Simulador de sistema de memoria de caches adaptativas con PIN.

Proyecto de Sistemas Informáticos.
Facultad de Informática, Universidad Complutense de Madrid

Profesor director: Manuel Prieto Matías
José Carlos Silva Cuevas
Sergio Carazo Alba
Rubén Nogales Cadenas

5 de julio de 2006

Índice general

Índice de figuras

Prefacio

El objetivo de nuestro proyecto es implementar un simulador dinámico de caches adaptativas. Con él podemos comprobar la eficacia de las técnicas de hardware adaptativo sobre diversos benchmarks. En primer lugar, se ha implementado un sistema de caches de datos e instrucciones con varios niveles, permitiendo configurar completamente las características de cada nivel. Tras esto, añadimos la posibilidad de adaptar dinámicamente el número de vías en función de la tasa de fallos. Para instrumentar dinámicamente el código hemos utilizado la herramienta Pin desarrollada por Intel. Como era necesario validar los resultados obtenidos, los hemos comparado con los obtenidos por otro simulador de caches, Dinero IV. Las pruebas demuestran que el error entre unos y otros es muy reducido. Por último se llevaron a cabo pruebas para estudiar la eficacia de los mecanismos adaptativos en cache. Los resultados obtenidos demuestran que, sin incrementar significativamente la tasa de fallos, sí se consigue reducir el número de vías de cada nivel. Esto supone una mejora en el consumo energético de la jerarquía de memoria.

English version

The goal of our project is to develop an adaptative cache dynamic simulator. We can use it to test the effectiveness of adaptative hardware techniques in several benchmarks. First, we have built a shared multilevel cache. It is possible to configure completely each level modifying its configuration file. After that, we add the possibility of dynamically adapt the number of ways according to miss rate. To dynamically instrument code we have used Pin, property of Intel. In the way to validate results, we have compared our memory system with Dinero IV, another cache simulator. The comparative demonstrate results are very close. Finally, tests were carried out to study the effectiveness of adaptative methods applied in caches. Results demonstrate that, without increasing miss rate significantly, it is possible to reduce the average number of ways each level has. This involves an improvement in the power consumption of the memory hierarchy.

Capítulo 1
Introducción

1.1. Contexto del proyecto

La continua evolución en pos de un mayor rendimiento, unida a la cada vez mayor disponibilidad de transistores, ha provocado la incorporación de elementos cada vez más complejos dentro del procesador, muchos de los cuales sólo son beneficiosos bajo circunstancias muy particulares. Esta tendencia, ha hecho que para ciertos programas, una gran parte de los recursos son innecesarios o están infrautilizados, y su uso comporta un desperdicio innecesario de energía, con los problemas que ello conlleva. Con el propósito de evitar este malgasto de energía, han surgido recientemente diversas técnicas que permiten variar dinámicamente la configuración del procesador para adaptarlo a las necesidades específicas del programa o fragmento de programa en curso. Dichas técnicas pueden agruparse bajo la denominación genérica de "Adaptación Dinámica de Estructuras del Procesador" o bien, de un modo más general, podemos referirnos a ellas como técnicas de "Gestión Eficiente de Recursos del Procesador". Los beneficios que pueden conseguirse con este paradigma son evidentes: ejecutar la aplicación con el hardware justo que precisa puede mejorar el consumo de energía. Sin embargo, para asegurar un buen comportamiento de estas técnicas, es necesario que la sobrecarga introducida sea razonable. Además, la pérdida de rendimiento debida a la reducción de los recursos debe estar dentro de unos límites aceptables. En general, el diseñador debe fijar un compromiso entre la energía ahorrada y el deterioro en el rendimiento.

En relación al sistema de memoria cache, el hardware adaptativo conlleva una serie de complejidades inherentes al tipo de actividades que realizan. Por ejemplo, es preciso señalar que cuando se lleva a cabo la desactivación de un sub-array, previamente hay que salvar su información en el nivel superior de la jerarquía de memoria. De lo contrario, la semántica de los datos podría verse comprometida. No obstante, la penalización que esto conlleva no es sig-

nificativa si los cambios de configuración son poco frecuentes. El mecanismo de control de la adaptación debe por tanto garantizar esta circunstancia

Existen algunos simuladores importantes de sistemas cache como por ejemplo DINERO, la gran diferencia de nuestro proyecto, a parte de que el nuestro implementa la posibilidad de simular hardware adaptativo, es que DINERO es un simulador estático, esto es, es necesario generar previamente la traza del programa para que después DINERO la analice y devuelva los resultados. Los tamaños de dichas trazas pueden ir desde varios megabytes hasta incluso terabytes, lo que hace impracticable el uso de DINERO en muchos casos. Nuestro simulador sin embargo es dinámico, es decir, simula el comportamiento de la memoria conforme se ejecuta el mismo, sin necesidad de generar ningún fichero de traza.

1.2. Objetivos del proyecto

Las tareas realizadas han consistido en la investigación sobre jerarquías de memoria adaptativas orientadas a la reducción del consumo de energía en las memorias cache. Dentro de este contexto, las propuestas más relevantes que han aparecido en la literatura ha aprovechado la división en sub-arrays de este tipo de estructuras, con el objeto de desactivar las partes que no sean necesarias. En particular, se han presentado esquemas que permite variar dinámicamente tanto la asociatividad como el número de conjuntos que la integran, mediante la desactivación (gating) de los sub-arrays que forman la(s) vía(s) o conjuntos. El principal handicap que presentan estos diseños se debe al hecho de que para poder desactivar un sub-array es preciso salvar previamente su información en el nivel superior de la jerarquía de memoria (operaciones de salvaguarda). De lo contrario, la semántica de los datos podría verse comprometida.

En este proyecto hemos desarrollado un simulador de una jerarquía de memoria adaptativa que permite variar tanto la asociatividad, como el número de conjuntos. El objetivo final perseguido es inferir el comportamiento de este tipo de hardware, evaluar la sobrecarga inducida por las operaciones salvaguarda y analizar heurísticas de y se podrá arrojar luz sobre la eficiencia de este tipo de técnicas.

Para poder llevar a cabo el proyecto hemos optado por capturar dinámicamente las instrucciones de los procesos en tiempo de ejecución, a fin de desarrollar un simulador dinámico. Para ello hemos hecho uso de PIN, una herramienta de optimización dinámica desarrollada recientemente por Intel. Los simuladores convencionales, como DINERO IV, requieren la generación previa de trazas. Este hecho platea serios inconvenientes en aplicaciones reales

ya que las trazas pueden alcanzar un tamaño del orden de Gigabytes o incluso Terabytes, lo que limita seriamente el tipo de benchmarks que pueden ser evaluados. Asimismo, un simulador dinámico puede ser extendido con más facilidad para evaluar el impacto de la concurrencia en sistemas multiprocesador.

En primer lugar será necesario diseñar toda la estructura que nos permita simular el comportamiento normal de un nivel de cache y añadir las características que hacen de la cache que sea adaptativa, es decir, implementar la posibilidad de que transforme su tamaño o su asociatividad en tiempo de ejecución. Una vez hecho esto, hay que extenderlo a una jerarquía de memoria compuesta por varias caches. Y el último paso consistirá en hacer uso de PIN convenientemente para poder realizar las llamadas oportunas durante la ejecución del programa.

1.3. Guía de capítulos

Como se ha descrito antes, este proyecto está relacionado con distintos temas que es necesario tratar más en profundidad para entender su diseño. Lo haremos en los siguientes capítulos de esta manera:

- En el capítulo 2 daremos una visión del hardware adaptativo. Explicaremos algunas de las características más importantes, el tipo de estructuras adaptativas que podemos encontrarnos y los mecanismos que se utilizan para implementarlos.

- En el capítulo 3 trataremos la herramienta PIN. Hablaremos de su funcionamiento, su utilización, sus características y cómo está diseñada e implementada. Esto nos servirá para poder entender mejor el diseño de nuestro simulador.

- En el capítulo 4 pasaremos a explicar este diseño y su posterior implementación.

- Por último, el capítulo 5 estará dedicado a las pruebas realizadas sobre varios SPEC. Hablaremos de los SPEC utilizados, los resultados obtenidos en las pruebas y las conclusiones que se puedan inferir de ellos.

Capítulo 2
Hardware adaptativo

Empezaremos este capítulo realizando una introducción sobre la necesidad del uso de técnicas adaptativas (sección 2.1.), incidiendo en la variabilidad del comportamiento de las aplicaciones en un mismo sistema hardware (subsección 2.1.1.), las estructuras hardware propensas a ser objeto de adaptación (subsección 2.1.2.) y cómo y cuándo será necesario adaptar el hardware (subsección 2.1.3.).

En la sección 2.2. examinaremos las distintas técnicas que se puede utilizar con el objeto de la adptavilidad, como serán la variación de la frecuencia y/o voltaje (subsección 2.2.1.) y diversas técnicas relacionadas con las memorias cache (subsección 2.2.2.).

2.1. Introducción a la sintonización del procesador para un procesamiento adaptativo

En esta sección justificaremos la conveniencia de la sintonización del hardware desde el punto de vista software - la variabilidad en la demanda de las aplicaciones- y describiremos en detalle los aspectos más relevantes que es necesario considerar para efectuarla -control de la adaptación, estructuras hardware, etc.-.

2.1.1. Variaciones en el comportamiento de las aplicaciones

Las necesidades de recursos hardware, como caches, colas de lanzamiento, lógica de búsqueda, etc., pueden variar mucho de una aplicación a otra, e incluso dentro de diferentes fases de una misma aplicación. Este comportamiento dinámico variable ha sido ampliamente demostrado por diversos

grupos de investigación a lo largo de los últimos años. Uno de los primeros estudios al respecto fue elaborado por el grupo del profesor David Albonesi [XuAl99]. En él compararon el grado de paralelismo exhibido por diferentes secuencias dinámicas de instrucciones extraídas de los SPEC95, observando cambios apreciables entre las mismas. Estas variaciones sugieren que el aprovechamiento de las estructuras del procesador, depende de la fase en la que se encuentre el programa: aquellas cuyo paralelismo sea elevado, se beneficiarán de estructuras de mayor tamaño, mientras que las fases con bajo paralelismo, obtendrán un rendimiento óptimo empleando menos recursos. Este comportamiento se aprovechó en beneficio de la reducción del consumo en múltiples trabajos posteriores, gracias a la aplicación de técnicas de adaptación dinámica de estructuras del procesador [BSB+00, BABD00, BaDA01, DKA+02, BKAB03, ChAD04].

Otro estudio sobre la variabilidad exhibida por los SPEC95 lo encontramos en [ShCa99]. En este trabajo, las aplicaciones se dividieron en intervalos, midiendo en cada uno de ellos una serie de parámetros como el IPC -Instrucciones por Ciclo-, los porcentajes de aciertos del predictor de saltos y del predictor de valores, el comportamiento de las caches, o la ocupación del buffer de reordenamiento. Los resultados demostraron la gran variabilidad de estos parámetros entre distintas fases del programa. Este estudio motivó posteriormente el desarrollo de una serie de herramientas (SimPoint) para mejorar la eficiencia de las simulaciones arquitectónicas [AuLE02] -reducir los tiempos de simulación sin afectar a la precisión de las mismas-.

2.1.2. Estructuras hardware adaptativas

Para explotar la variabilidad en la demanda de las aplicaciones mencionada en la sección anterior, es necesario seleccionar qué recursos pueden ser potencialmente objeto de adaptación -entendida como variación dinámica de su configuración- y cómo es posible efectuarla. En general, todas aquellas estructuras diseñadas de forma modular, como es el caso de las caches o las colas de instrucciones, son susceptibles de adaptación.

2.1.3. Control de la adaptación

Disponer de estructuras que sean susceptibles de adaptación no es suficiente para garantizar una ejecución más eficiente. Es necesario un mecanismo de control que tome la decisión de cuándo y cómo adaptar el hardware. Este control es esencial, ya que en general toda adaptación lleva asociadas ciertas penalizaciones que pueden llegar a ser muy perjudiciales si no se

mantienen por debajo de unos límites. El deterioro introducido puede limitarse exclusivamente al rendimiento, pero en casos extremos, una perdida de rendimiento -aumentar el tiempo de ejecución- , puede llegar incluso a afectar al consumo. El simple hecho de estar ejecutando durante más tiempo una aplicación, puede suponer un mayor consumo -debido básicamente a la distribución del reloj- , llegándose a anular las ganancias energéticas conseguidas con la adaptación. Se han propuesto técnicas tanto estáticas como dinámicas para el control de la adaptación. En el caso estático, la adaptación se realiza en base a un estudio previo de la aplicación -en tiempo de compilación o profiling- . En el caso dinámico, como su propio nombra indica, las decisiones se toman durante la ejecución del programa. Las técnicas estáticas tienen una visión global del programa, y requieren poco hardware adicional -básicamente, mecanismos para comunicar al hardware las decisiones a tomar-. Sin embargo, carecen de toda aquella información que sólo es posible conocer en tiempo de ejecución, y ello puede mermar su eficacia. Por su parte, las técnicas dinámicas toman la decisión a partir del comportamiento actual del programa, evitando este inconveniente a costa de un incremento de hardware. Podemos dividir el control de la adaptación en tres tareas principales. En primer lugar, es necesaria una monitorización para conocer la utilización de los recursos (1). En función de las estadísticas de utilización, se identifican las fases del programa (2), y por último se debe elegir la configuración más apropiada para cada una de ellas (3). A continuación detallaremos las funciones de cada una de estas tareas.

Monitorización del hardware adaptativo

Para guiar las decisiones de adaptación es necesario en primer lugar tomar medidas que permitan detectar fases a lo largo del programa y evaluar la efectividad de las adaptaciones efectuadas. En la literatura existen muchas propuestas tanto dinámicas como estáticas. Las propuestas estáticas [SPHC02, HuRT03, SaSk04], se suelen basar en un profiling previo de la aplicación, que ha de ser suficientemente representativo de lo que realmente ocurrirá al ejecutar la aplicación con datos reales. La mayoría de las propuestas que aparecen en la literatura son dinámicas y se basan en un muestreo periódico de contadores-hardware. En algunos casos es necesario añadir contadores específicos, mientras que en otros la monitorización se basa en los ya incluidos en la mayoría de los procesadores de propósito general. Ejemplos de contadores específicos los encontramos en [PoKG01, FoGo01] dos trabajos en los que se propone una lógica de lanzamiento adaptativa. La propuesta de Ponomarev et al. [PoKG01] estaba basada en la monitorización de contadores-hardware específicos que permitieran conocer el número de en-

tradas válidas de la ventana de instrucciones en un determinado instante, así como el número de ciclos en los que dicha ventana se había llenado. Alternativamente, Folegnani y González [FoGo01], registran para cada instrucción su posición en la ventana de instrucciones al comenzar su ejecución, y cuando se observa que no provienen del final de la ventana, se infiere que esta es demasiado grande. En [PYFV01] se propone una cache adaptativa en la que puede variarse el número de conjuntos que la integran. La estadística elegida en este caso para guiar la adaptación es el porcentaje de fallos en el acceso a la cache. Esta métrica puede calcularse fácilmente a partir de los contadores-hardware habituales en los procesadores actuales.

Identificación de las fases

Basándose en las estadísticas seleccionadas, el siguiente paso en la adaptación es identificar los cambios de fase que hacen necesario variar la configuración del procesador. Existen esencialmente dos alternativas: la aproximación temporal y la aproximación posicional. Ejemplos de la aproximación temporal los encontramos en [PoKG01, BABD00]. En general, la identificación de las fases, o más concretamente los cambios de fase, se detectan por la superación de un umbral o umbrales. La aproximación posicional se introdujo en [HuRT03]. A diferencia de las técnicas anteriores, en este caso se utiliza la propia estructura del código para identificar las fases. A partir de los datos obtenidos mediante profiling se delimitan las secciones de código más relevantes -entendiendo por relevante aquella sección que supere un cierto porcentaje del tiempo de ejecución total-, y se instrumenta el código de manera que se puedan identificar los puntos de entrada y salida de cada una de estas secciones.

Elección de la configuración

Por último, una vez definidas las fases, es necesario decidir la configuración más ventajosa en cada una de ellas. De nuevo, la decisión se puede tomar estática o dinámicamente. Una estrategia dinámica sencilla, propuesta en [BABD00], consiste en probar explícitamente las posibles configuraciones en tiempo de ejecución, y quedarse con la que mejor compromiso ofrezca. Obviamente, esto solamente es posible cuando el número de configuraciones alternativas es pequeño, y las sobrecargas o penalizaciones involucradas en los cambios de configuración son tolerables. La opción más habitual, sin embargo, consiste en identificar la configuración más apropiada en base a las estadísticas de monitorización seleccionadas. Por ejemplo, en [PoKG01], el tamaño de la ventana se elige de acuerdo a la ocupación media de dicha

ventana en el intervalo temporal anterior. Estas opciones asumen localidad temporal en las aplicaciones, es decir, suponen que el comportamiento del futuro próximo será similar al comportamiento exhibido en el pasado.

Alternativamente, la aproximación posicional [HuRT03], asume localidad espacial, es decir, supone que el comportamiento en las distintas invocaciones a una determinada sección de código se mantendrá constante a lo largo del tiempo. De ese modo, la elección de la configuración más ventajosa para una determinada fase se hará en función de las estadísticas obtenidas en instancias anteriores de esa misma fase -invocaciones anteriores de la correspondiente sección de código-.

2.2. Técnicas adaptativas

En las secciones previas hemos explicado las principales características de los sistemas adaptativos, describiendo con cierto detalle sus mecanismos de control. A continuación realizamos un repaso de las técnicas existentes, agrupándolas en función del elemento/parámetro del procesador objeto de adaptación.

2.2.1. Adaptación de frecuencia y voltaje

Para reducir el consumo de energía de un hardware síncrono, sin modificar su diseño, se puede actuar tanto sobre la señal de reloj como sobre la alimentación. Cuando un recurso no es necesario, es posible congelar la señal de reloj (clock gating), o desconectar la alimentación. En el primer caso se evitan conmutaciones de los transistores y por consiguiente se elimina la componente dinámica del consumo, mientras que en el segundo caso se eliminan ambas componentes (estática y dinámica). Por el contrario, si el recurso es necesario, pero no a pleno rendimiento, la frecuencia y la alimentación se pueden reducir siempre que el tiempo de cómputo no se vea afectado. Con ello se logra una reducción del consumo dinámico proporcional a la variación de estos parámetros.

Uno de los primeros en abordar la adaptación dinámica de la frecuencia de reloj fue Albonesi. En [Albo98], este investigador propuso aprovechar la fragmentación de ciertos elementos del procesador para realizar una adaptación de su tamaño y frecuencia. Sin entrar en detalles, la idea presentada en este trabajo consiste en reducir el tamaño de un elemento cuando este no se aprovecha en su totalidad, lo que permite aumentar su frecuencia de reloj y mejorar con ello su rendimiento. Si bien el objetivo de este primer trabajo se centraba en mejorar el tiempo por instrucción, esta idea tam-

bién puede ser aplicable al consumo, como de hecho hizo el propio autor en trabajos posteriores. En [SMB+02, MSS+03] se propone dividir el procesador en varios dominios de reloj (MCD: Multi-Clock Domain), cada uno de los cuales utiliza la tensión de alimentación y la frecuencia de reloj mínimos que garantizan un rendimiento adecuado. El control de ambos parámetros se lleva a cabo mediante profiling. Según sus experimentos, esta solución consigue unos ahorros de energía significativos con pequeñas pérdidas de rendimiento. La evolución de estas propuestas que acabamos de mencionar, que emplean dominios de reloj independientes, pasa por los procesadores globalmente-asíncronos/localmente-síncronos (GALS: Globally Asynchronous Locally Synchronous) [MTC+03, IyMa02, DSA+04].

Propuestas similares aparecen en estudios como [HsKH01, Marc00, ChTM01, BPSB00, PeBB00, GLF+00, PoLS01, FlRM02, XiMM03]. En los dos primeros, el control del voltaje y la frecuencia se realiza dinámicamente en función de los fallos de cache, mientras que en [ChTM01] la métrica empleada es el IPC. En los restantes trabajos este control se lleva a cabo con la ayuda del sistema operativo o del compilador. Dentro de esta tendencia de diseño, una de las propuestas más novedosas consiste en aplicar técnicas de la teoría de control, con el fin de compensar la velocidad de las distintas etapas del procesador [JWP+05, WJM+05].

Un diseño algo diferente a los que acabamos de describir, se presenta en [BaMo02]. El procesador incorpora dos caminos de ejecución independientes, uno de elevado rendimiento y consumo, y el otro de menor consumo pero rendimiento inferior (siendo la frecuencia de reloj y el voltaje de alimentación de este último la mitad del primero). El objetivo de este diseño es ejecutar las instrucciones críticas en el camino de alto rendimiento, y el resto en el de bajo consumo.

2.2.2. Adaptación de las memorias cache

El consumo de energía de las memorias cache representa una parte muy significativa del consumo total del chip (debido tanto a su elevado número de accesos como a su área), y por este motivo se han dedicado importantes esfuerzos a reducirlo. Muchas de las propuestas hacen uso de técnicas de adaptación dinámicas. A continuación describiremos con cierto detalle las más significativas, algunas de las cuales servirán de base en los mecanismos de adaptación desarrollados en este trabajo.

Una aproximación arquitectónica, aunque no adaptativa, consiste en incorporar una pequeña cache, denominada cache de filtrado (filter cache), para reducir el número de accesos al primer nivel y de este modo reducir su energía dinámica [KiGM97, KiGM00].

Otra posibilidad consiste en reducir el tamaño del primer nivel de cache. Para ello, se aprovecha la división en sub-arrays de esta memoria, con el objeto de desactivar las partes que no sean necesarias. La forma más sencilla de proceder, sería basar la decisión en la dirección del acceso, y activar sólo el sub-array que contiene el dato. No obstante, esta aproximación conlleva un incremento de la latencia, al tener que secuenciar las tareas de selección y acceso. Para evitar esta penalización, las soluciones más frecuentes toman la decisión empleando adaptación dinámica.

En [Albo00], se presenta un esquema que permite variar dinámicamente la asociatividad, mediante la desactivación (gating) de los sub-arrays que forman la(s) vía(s). En este caso, la gestión de la adaptación se realiza con la ayuda del software. Evidentemente, esta técnica sólo se puede aplicar si la cache es asociativa por conjuntos, no obstante esto no supone un gran inconveniente ya que se trata del caso habitual. Al igual que sucede con el resto de técnicas adaptativas, es necesario tener cautela al reducir la asociatividad, pues de lo contrario el rendimiento y el consumo pueden verse seriamente penalizados.

Los elementos necesarios para llevar a cabo la gestión dinámica de la asociatividad de la cache son los siguientes:

- Tener las vías distribuidas en sub-arrays, algo que por motivos de latencia ocurre en la mayor parte de los casos.
- Hardware adicional para la deshabilitación (gating) de sub-arrays.
- Un conjunto de instrucciones especiales.
- Soporte para salvar el contenido de los sub-arrays desactivados, y mantener la coherencia cache.

Las figuras 2.1 y 2.2 muestran respectivamente un ejemplo de organización de cache asociativa por conjuntos (4 vías), y la estructura de los sub-arrays. La Figura 2.3 ilustra las modificaciones que es necesario introducir para llevar a cabo la desactivación (gating) de un sub-array. Empleando este diseño, cuando la línea Activación toma el valor cero, se deshabilita la precarga del sub-array, con lo que no se seleccionará ninguna wordline, y prácticamente no se consumirá energía dinámica. Gracias a esta modificación, en este caso es posible emplear tres configuraciones distintas, cada una de ellas con el mismo número de conjuntos, pero distinto grado de asociatividad (1, 2, 4). A modo de ejemplo, en la Figura 2.1 se han resaltado en gris los sub-arrays desactivados para la configuración de dos vías.

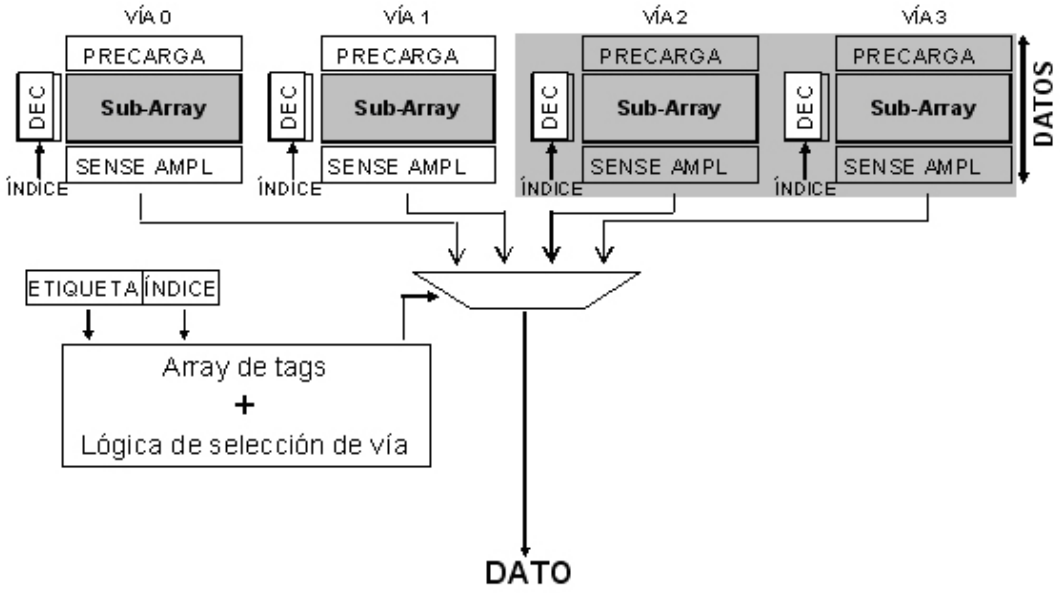

Figura 2.1: Ejemplo de organización de una cache asociativa por conjuntos de 4 vías

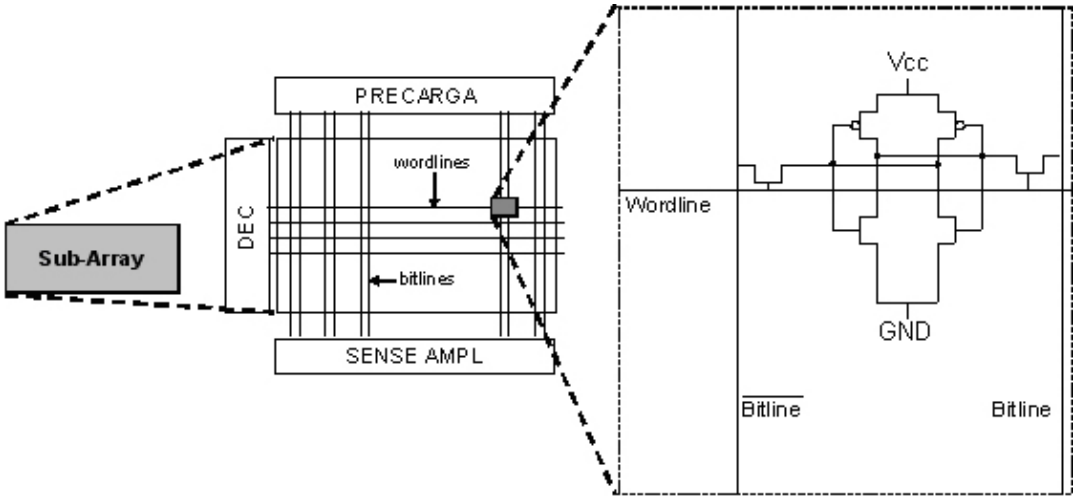

Figura 2.2: Estructura de un subarray

Si las vías a su vez se dividen en sub-arrays, cabe la posibilidad de aplicar esta misma idea para la desactivación de conjuntos, como de hecho han propuesto algunos autores [YPFV02]. En la Figura 2.4 se muestra un ejemplo de organización de cache de este estilo y en ella se han resaltado los sub-arrays

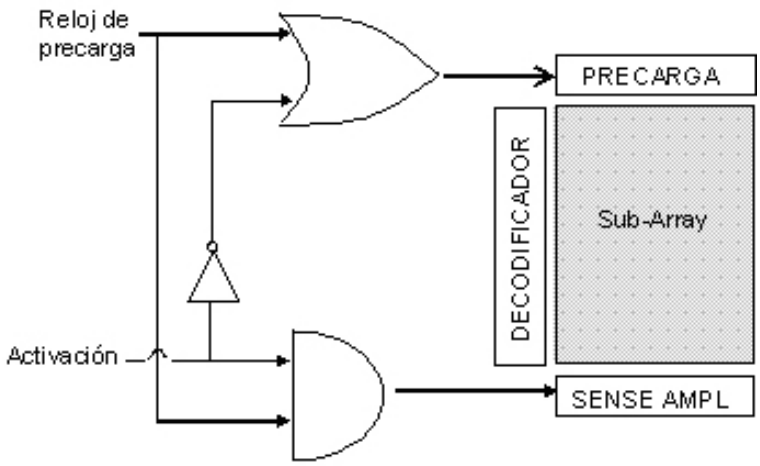

Figura 2.3: Modificación de un subarray para permitir su desactivación (gating)

que serían deshabilitados para reducir el número de conjuntos a la mitad. La única diferencia con respecto al caso anterior radica en la necesidad de variar el tamaño efectivo de la etiqueta y del índice. En la práctica esto se soluciona fijando el tamaño de la etiqueta al máximo posible, y variando el índice según el número de sub-arrays activos, como queda reflejado en la Figura 2.5. Es obvio que esta manipulación implica un ligero sobrecoste hardware respecto a la propuesta de Albonesi, sin embargo, permite una mayor granularidad en la adaptación de la cache. En [YPFV02] se han considerado también propuestas híbridas aplicando ambas técnicas.

La desactivación de sub-arrays empleando gating permite reducir únicamente la energía dinámica. Para reducir también la componente estática es necesario, además, modificar el diseño de las celdas tal y como propusieron Yang et al. en [YPF+01]. La Figura 2.6 ilustra el esquema de una celda de memoria estática convencional, así como la alternativa de Yang. Esta solución ha sido adoptada recientemente por Intel en el diseño del procesador Yonah [Krew05], para la adaptación de la asociatividad de la cache. Una posible mejora a este esquema consiste en evitar la pérdida de los datos, y por consiguiente no tener que salvarlos antes de la desactivación. En [AgLR02] se propone un diseño mejorado del transistor de desactivación que permite lograr este objetivo.

Otros trabajos similares son [KFBM02, KaHM01, SaSk04, KFBM04], en las que la deshabilitación no apaga totalmente el conjunto, sino que lo alimenta con una tensión menor. De esta forma es posible lograr que no se pierdan los datos, y al mismo tiempo reducir casi completamente la energía estática

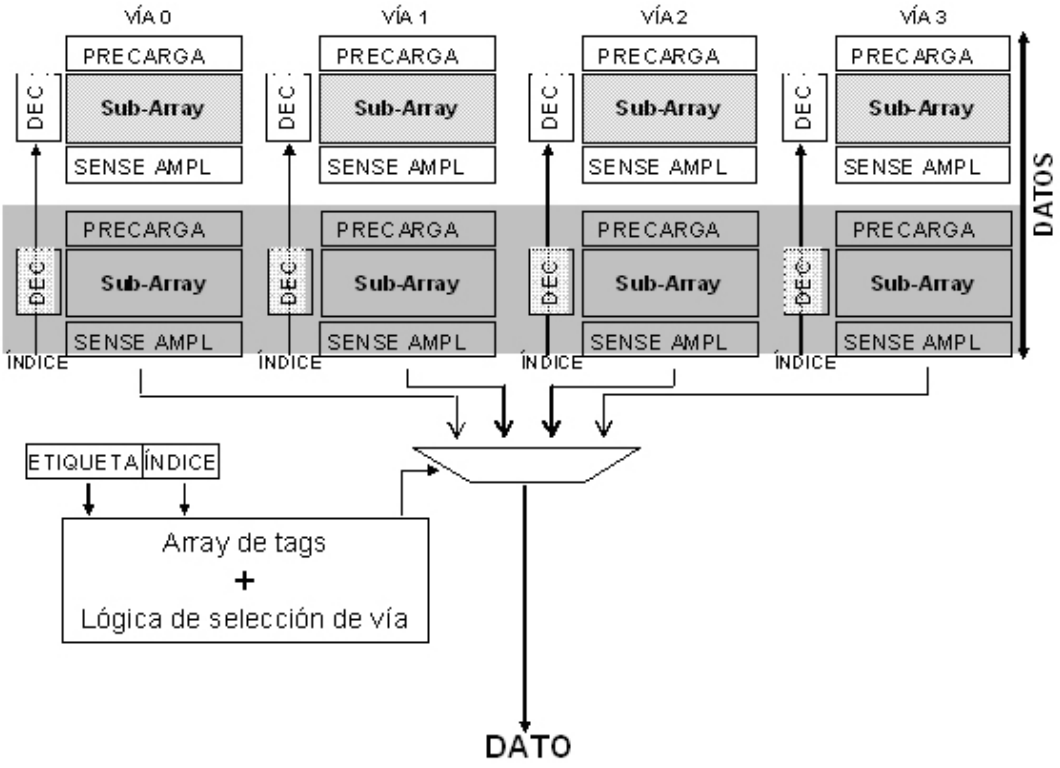

Figura 2.4: Variación del número de conjuntos activos en la cache, de acuerdo a lo propuesto en [YPFV02].

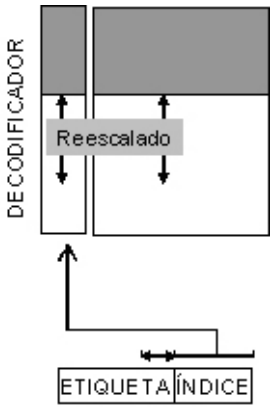

Figura 2.5: Variación del tamaño del índice y de la etiqueta, de acuerdo a lo propuesto en [YPFV02].

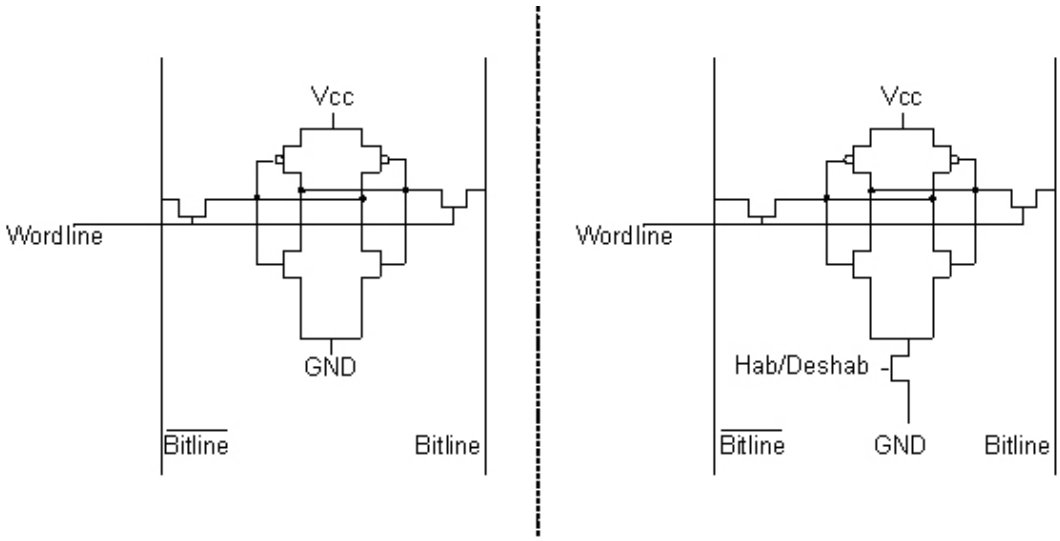

Figura 2.6: Celda de memoria cache (SRAM): (a) versión original, (b) versión Gated-$_{Vdd}$, propuesta por Yang et al. en [YPF+01].

cuando se encuentra en el estado deshabilitado. Por último, una serie de esquemas relacionados también con la adaptación de caches los encontramos en [DBB+02, XHD+02, ZaVN03, DhSm02].

Capítulo 3
La herramienta PIN

En este capítulo trataremos de dar una visión acerca de la herramienta desarrollada por Intel, PIN, y utilizada en el desarrollo de nuestro simulador. PIN es un motor que reescribe código dinámicamente, ya sea de programas alojados en memoria o incluso librerías, pero sin modificar la fuente original, sino copiando el código ubicado en la cache de instrucciones e insertando sus propias llamadas.

En primer lugar veremos sus características y hablaremos del concepto de instrumentación, esencial en el desarrollo del proyecto. Tras esta introducción explicaremos cómo funciona la herramienta y después analizaremos su diseño e implementación.

3.1. Introducción

La instrumentación de código para su monitorización se puede aplicar en diferentes fases: directamente en el código fuente, en tiempo de compilación, después del enlazado, o durante la ejecución. Pin es una herramienta que instrumenta de esta última manera. Está inspirado en ATOM, en el sentido de que el usuario puede instar llamadas a las rutinas de instrumentación en cualquier lugar del ejecutable. Asimismo, Pin proporciona una extensa API para conseguir unas pintools portables, aunque también se incluyan instrucciones para acceder a los niveles más bajos de la arquitectura. Como Pin instrumenta durante la ejecución, necesita optimizar al máximo la inserción de código. Para ello utiliza caches de código, inlining, planificación de instrucciones, renombramiento de registros, etc.; con lo que conseguimos automatizar la optimización, en lugar de delegar esta responsabilidad al usuario. Pin conserva las direcciones de memoria (instrucciones y datos) y valores (registros y memoria) que tendría el programa si no estuviese instrumentado, así mantenemos el comportamiento del programa original. Además Pin, como si de un depurador se tratase, puede instrumentar el código de un pro-

grama que esté en ejecución, o dejar temporalmente de hacerlo, lo que es muy útil en cuando las aplicaciones requieren mucho tiempo.

3.2. Instrumentando

El API de Pin permite conocer el estado completo de un proceso, como por ejemplo el contenido de los registros, memoria y el flujo de control. El usuario añade procedimientos al proceso (conocidas como *rutinas de análisis*), y escribe *rutinas de instrumentación* para determinar en qué momento serán llamadas esas rutinas de análisis. Los parámetros de las rutinas de análisis pueden ser variables de estado o constantes. Pin también tiene la capacidad (limitada) de alterar el comportamiento de un proceso sobreescribiendo en los registros o en la memoria del proceso a través de una rutina de análisis.

La instrumentación se ejecuta mediante un compilador dinámico (compilador just-in-time o simplemente JIT). La entrada de este compilador es el propio código del ejecutable original. Pin intercepta la ejecución de la primera instrucción del ejecutable y genera ("compila") una nueva secuencia con código nuevo ya instrumentado, que comienza en esa misma instrucción. Entonces transfiere el control a la secuencia recién generada. Este nuevo código es casi idéntico al original, pero Pin inserta el código necesario para recuperar el control en cuanto aparezca un nuevo salto. Cuando recupere el control, Pin generará más código para el destino del salto y continuará la ejecución. Este nuevo código y su instrumentación se guardan en una cache de código para una futura ejecución de la misma secuencia de instrucciones, con el fin de mejorar el rendimiento.

Por ejemplo, con la instrucción de la API de Pin $INS\_AddInstrumentFunction(funcion, \_)$ conseguimos que Pin intercepte cada instrucción del programa original, y se llame a función. En lugar de interceptar cada instrucción, podemos trabajar con la trazas ($TRACE\_AddInstrumentFunction(traza, \_)$), con bloques básicos ($BBL\_AddInstrumentFunction(bbl, \_)$), incluso con imágenes enteras de un proceso. En este sentido Pin provee un API bastante extenso para la inspección e instrumentación de código. En el apéndice B hemos incluido el manual de referencia de PIN con una completa descripción del API.

3.3. Diseño e implementación

En esta sección comenzaremos comentando un esquema general de Pin y su funcionamiento. Después explicaremos de qué manera se hace con el control de la aplicación y por último el proceso mediante el cual compila dinámicamente la aplicación y la instrumenta.

3.3.1. Esquema del sistema

La figura 3.1 ilustra la arquitectura de Pin. En el nivel de abstracción más alto PIN consiste en una máquina virtual (VM), una cache de código y una API de instrumentación que es invocada por las Pintools. La máquina virtual está compuesta por un compilador JIT (just-in-time), un emulador y un despachador. Una vez que Pin se hace con el control de la aplicación, la máquina virtual coordina todos sus componentes para ejecutar la aplicación. El JIT compila e instrumenta el código de la aplicación y el despachador lo lanza. El código compilado se almacena en la cache de código, y el emulador interpreta las instrucciones que no se pueden ejecutar directamente. Se usa en llamadas a sistema que requieren un manejador especial para la VM. Como Pin se sitúa por encima del sistema operativo, sólo puede capturar instrucciones a nivel de usuario.

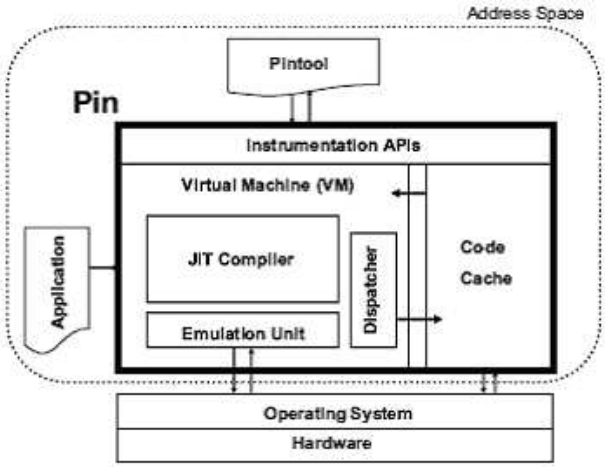

Figura 3.1: Arquitectura software de Pin

Se puede observar también que hay tres programas presentes cuando el proceso instrumentado está ejecutándose: Pin, la pintool y el propio ejecutable. Pin es el motor que permite compilar dinámicamente e instrumentar la aplicación. La Pintool contiene la instrumentación y las rutinas de análisis

y se enlaza con una librería que le permite comunicarse con Pin. Mientras compartan el mismo espacio de direcciones no podrán compartir ninguna librería, y es por eso se crean 3 copias de la librería glibc. Haciendo que el uso de estas librerías sea privado, una para cada uno de estos 3 programas, evitaremos que haya alguna interacción no deseada entre ellos.

3.3.2. Arrancando PIN

Pin se carga en el mismo espacio de direcciones que la aplicación mediante la API de Unix Ptrace, con la que se hace con el control de la aplicación y captura el contexto del proceso. Inicia la Pintool, que tras cargarse también en el mismo espacio de direcciones, devuelve el control a Pin. Pin que crea el contexto inicial y comienza con la compilación dinámica en el punto de entrada. Usando Ptrace podemos instrumentar un proceso que ya se esté ejecutando, como si estuviésemos trabajando con un depurador. También es posible parar la instrumentación de un proceso y continuar con la ejecución original, sin instrumentar.

3.3.3. El compilador JIT

El compilador genera código directamente con el mismo formato de instrucciones que el código fuente, sin usar un formato intermedio, y lo coloca en la cache de código. Sólo se ejecuta el código ubicado en la cache de código, nunca el original. La compilación se realiza traza a traza, entendiendo como tal la secuencia de instrucciones que termina en un salto incondicional o tras un número predeterminado de instrucciones). Cada traza puede tener múltiples puntos de salida, y cada salida apunta a un stub, que se encarga de redirigir el control hacia la máquina virtual. La VM determina la dirección destino y se encarga de generar una nueva traza a partir de ella, si es que no ha sido generada previamente. Después continua la ejecución de esta traza en la dirección destino.

En el resto de la sección discutiremos los siguientes puntos relacionados con el compilador JIT: trace linking, renombramiento de registros y la optimización de la instrumentación.

Trace Linking

Para incrementar el rendimiento, Pin intenta saltar directamente desde una salida de la traza hasta el destino del salto, obviando el stub y la máquina virtual. Este proceso es llamado trace linking. Cuando sólo hay un posible destino simplemente se enlaza el final de una traza con la dirección destino.

Sin embargo, es posible (debido a saltos, llamadas o returns) que haya más de un posible punto de destino. Es necesario algún tipo de predicción de destino, para los saltos indirectos.

La figura 3.2 ilustra como se lleva a cabo el trace linking cuando hay más de un posible destino, tal y como está implementado en las arquitecturas x86. Pin traduce un salto indirecto por un MOVE y un salto directo. Se almacena la dirección destino en el registro %edx y se van saltando a las direcciones almacenadas en la cache de código, hasta que ambas direcciones coincidan, entonces se continua con la ejecución de la traza. Si ninguna coincide se salta a LookupHtab1 que busca la dirección en una tabla Hash. Si se tiene éxito, se salta a la dirección ("traducida") que indique dicha tabla, sino se pasa el control a VM para que resuelva el salto indirecto.

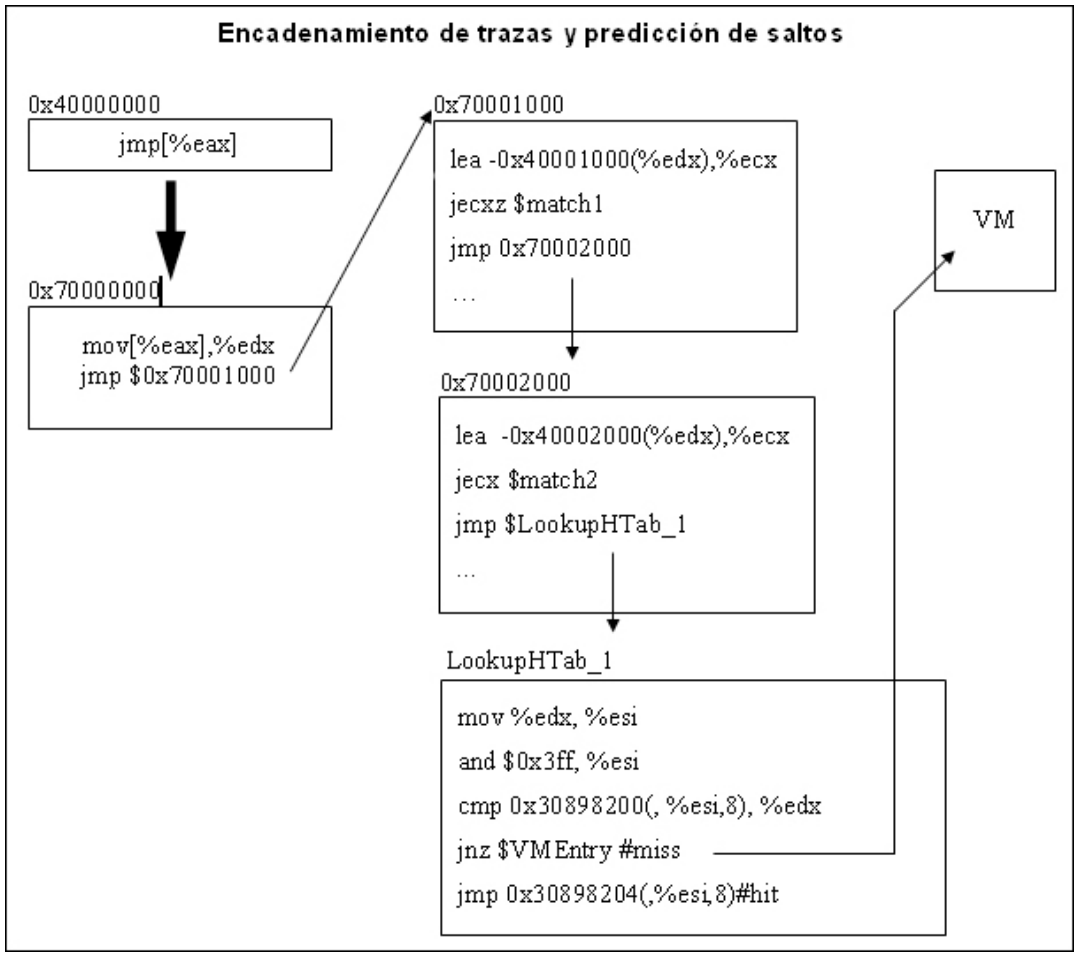

Figura 3.2: Compilando saltos indirectos y returns

Renombramiento de registros

Durante la compilación, Pin necesita registros libres, por ejemplo los que se ha visto que se usan en la figura 3.2. Cuando la instrumentación inserta una llamada en un proceso, el compilador debe de asegurarse de que no se sobreescribirá ningún registro que se esté utilizando en esos momentos. Con este fin, Pin reubica los registros utilizando una búsqueda lineal. Para realizar esta asignación de registros libres necesitamos saber, que registros están siendo usados (register liverness analysis), guardando el estado en una tabla hash con clave la dirección de la traza procesada (para realizar el proceso de una manera incremental). El JIT también debe asegurarse que la asignación de los registros a la salida de la traza origen debe coincidir con la entrada de la traza destino (reconciliation of register bindings). La figura 4.3 muestra como Pin puede reubicar los registros para el código original

Thread-local Register Spilling

Pin reserva un área ("spilling area") de memoria para registros virtuales (por ejemplo EAX y EBX de la figura 3.3b entrarían dentro de este "spilling area"). Cuando Pin comienza la ejecución de un hilo, crea el spilling area para este hilo y se hace con un registro físico para que sea su "spill pointer" que apunte a este área. Cada vez que se quiera acceder al spilling area sólo habrá que acceder a su spill pointer. Pin por defecto asume que el proceso que intercepta posee un único hilo, si se percata de que es multihilo (puede hacerlo ya que intercepta todas las llamadas al sistema de creación de hilos), invalida la cache de código y recompila la aplicación haciendo uso del spill pointer para acceder al spilling area correspondiente.

3.3.4. Optimizando el rendimiento de la instrumentación

Los desarrolladores de PIN comprobaron que la mayor parte de la sobrecarga que produce la instrumentación es causada por el código de instrumentación, en lugar de ser por el tiempo de compilación. Por tanto, conviene utilizar parte del tiempo de compilación en optimizar las rutinas de análisis. Por supuesto, esta optimización dependerá de la índole de la rutina de análisis, si ésta es muy compleja no se podrá apenas optimizar. Existen alguno métodos de optimización como por ejemplo, utilizar "inlining", para reducir así el número de llamadas a funciones u otras rutinas.

Otra optimización se llevó a cabo mediante el "liveness analysis" del registro *eflags*, se vio que dicho registro no era necesario cada vez que se

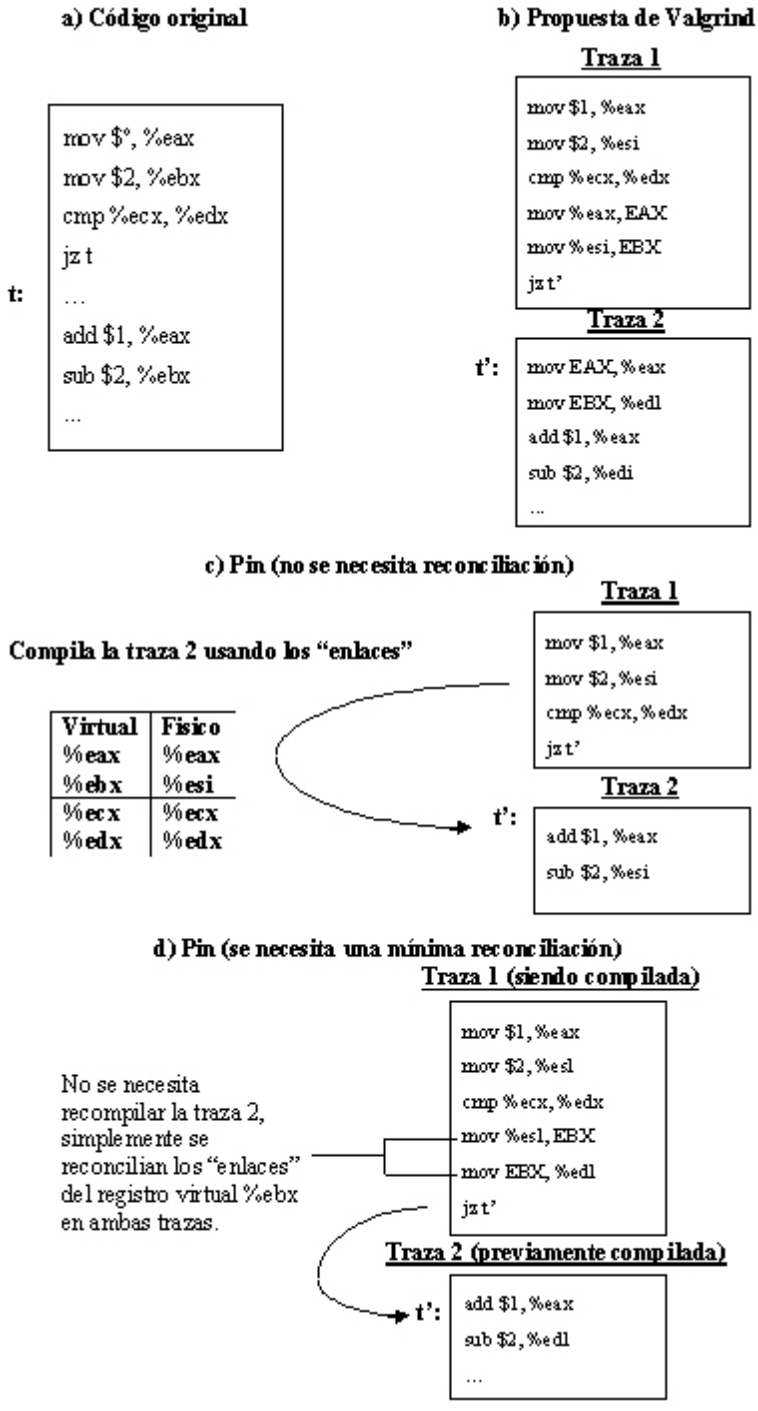

Figura 3.3: Reconciliation of register bindings

insertaba una rutina de análisis y dejó de guardarse su estado para restaurarlo más tarde (lo que suponía una perdida de rendimiento).

Por último, para conseguir una mayor eficiencia, el programador de Pintools puede reducir el número de puntos de inserción de código haciendo uso del API para acceder a trazas y bloques en lugar de hacerlo en cada instrucción.

Capítulo 4
Implementación de un Simulador de Caches Adaptativas basado en Pin

La implementación de la simulación se ha realizado en C++; debido a que las pintools deben estar descritas en el mismo lenguaje que PIN.

En el presente proyecto se han desarrollado 3 pintools, cada una implementando un tipo de cache distinto.

En primer lugar, implementamos una cache de instrucciones, después una cache de datos, y después integramos ambas en una jerarquía compuesta de dos niveles: una cache de instrucciones, otra de datos, y una cache compartida, a la que se accede en caso de que no se encuentre el dato o la dirección en los niveles inferiores.

Todos estos tipos de caches se encuentran implementados en sendas pintools, que tras capturar todas y cada una de las instrucciones, delegan la gestión de los datos a los archivos fuente que comentaremos a continuación.

Todos estos sistemas cache tienen el añadido de que se puede variar su funcionamiento en tiempo de ejecución. Se establece un rango de tasa de aciertos en el que se deben mover cada una de estas caches. Si están por debajo de un umbral, incrementaremos el número de vías (o aumentaremos el número de conjuntos) para mejorar el rendimiento, mientras, que si está por encima de otro umbral, se reducirá el número de vías (o se rebajará el número de conjuntos) con motivo de minimizar el uso de las estructuras cache.

También mostramos el diagrama de dependencias de compilación de los archivos fuentes utilizados en el proyecto (Fig 4.1).

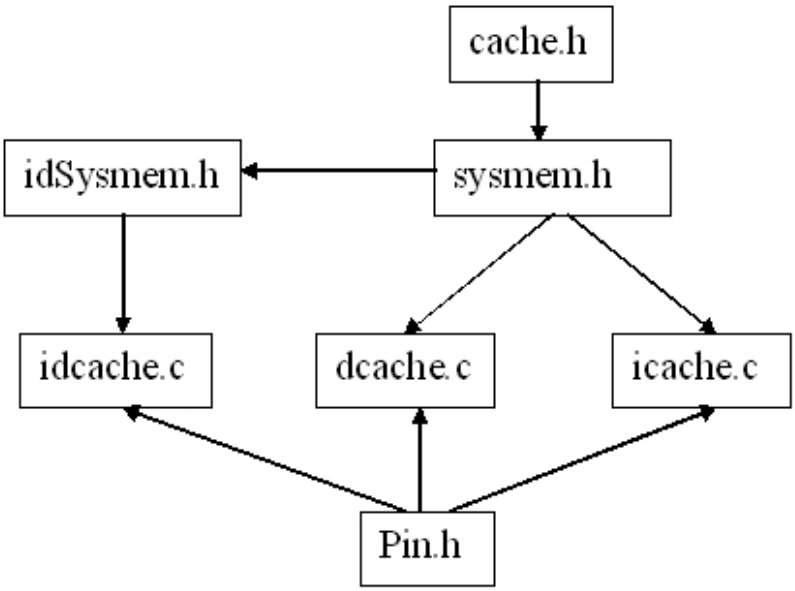

Figura 4.1: Grafo de dependencias de compilación de los fuentes del proyecto

4.1. Pintools

Todas las pintools implementadas (cache de instrucciones, de datos y mixta) comparten algunas partes de su código que detallaremos a continuación:

- Para poder indicar mediante parámetros por la línea de comandos, donde queremos que se ubique la salida que produce la pintool, además de para establecer el periodo con el que se realizará el muestreo utilizamos 2 objetos KNOB, cuyos valores por defecto están establecidos a 1000 para el caso del periodo, y a "dcache.out", "icache.out" o "idcache.out" para el caso de la salida de la cache de datos, de instrucciones o de instrucciones y datos, respectivamente.

- Una función de finalización, que se encarga de volcar al archivo de salida todas las estadísticas que se han ido recogiendo durante la ejecución del programa asociado.

- Para poder variar las características de la cache simulada sin tener que

volver a compilar otra vez la pintool, las características de la jerarquía de memoria se lee directamente de archivos ".in", según sea el caso.

También es necesario decir que cada una de las caches siguientes tiene varios niveles fijados por el ".in". La jerarquía de memoria gestiona los accesos empezando por el nivel 1, si en este no se encuentra el dato, pasaremos al siguiente, y así sucesivamente, hasta que se encuentre en algún nivel (si se halla de hecho en el sistema de memoria, si no se producirá un fallo de cache). A partir de aquí, las 3 caches difieren en el tratamiento que realizan de cada instrucción capturada, y en la implementación de las instrucciones auxiliares necesarias implementación:

4.1.1. Cache de instrucciones

Se captura la instrucción, y según sea de tamaño simple (4 bytes) o mayor, se realiza un "acceso simple" o un "acceso múltiple", cuyo significado ya se comentará más adelante.

En esta pintool también se ha incorporado la posibilidad de, en lugar de tomar las direcciones dinámicamente desde Pin, extraerlas de un archivo de trazas en formato Dinero, que haya sido generado previamente de de una manera estática.

4.1.2. Cache de datos

En este caso se captura la instrucción, y procedemos a comprobar si la instrucción es de lectura o de escritura en memoria.

En el caso de la lectura de dato tomaremos el tamaño de la lectura que se vaya a realizar, y viendo si el tamaño es simple o múltiple haremos una "carga simple" o "múltiple", que tendrán básicamente el mismo significado que el "acceso simple" y el "acceso múltiple" anterior en la cache de instrucciones. Si es una instrucción de escritura, realizaremos un "almacenamiento simple" o "múltiple"

4.1.3. Cache mixta

Esta pintool funciona básicamente como la cache de datos y de instrucciones juntas en un nivel1, siguiéndolas a continuación en el nivel 2 una cache compartida de datos y de instrucciones, por lo que cualquier fallo en los datos o instrucciones produce un acceso en la cache compartida. Por tanto, admite todo tipo de cargas, escrituras y accesos.

4.2. Implementación de una cache

Para empezar hay que definir las etiquetas, que en el caso multinivel que se trata pueden tener distintos estados, a saber: inválido, válido y modificado. Cada etiqueta está identificada por un número entero, Tag.

A continuación se definen los conjuntos, que, al ser los más utilizados en la prácticas y el el caso práctico que se está simulando, serán asociativos por conjuntos. Cada conjunto también permite una asociatividad máxima y mínima (este es el rango en el que trabajará la adaptabilidad). También serán necesarios acumuladores que me indiquen el elemento siguiente a eliminar, ya que se está trabajando con una política de emplazamiento Round Robin.

Seguidamente, ya se precisa el esqueleto de una cache, especificando el tamaño total, el de cada marco de bloque y la asociatividad inicial, que variará según se vayan cambiando el número de conjuntos.

También hay que concretar el número máximo de conjuntos que podrá tener la cache, así como la política de escritura en memoria ("escritura inmediata" o "post-escritura"). "Escritura inmediata" significa que cuando se actualiza un bloque en la memoria cache debe actualizarse igualmente en la memoria principal, mientras que "post-escritura" quiere decir que la escritura en memoria principal se realizará cuando se produzca el desalojamiento del bloque correspondiente de la cache, según dicte la política de emplazamiento.

El corazón de la cache es la que se encarga de gestionar los accesos tanto simples como múltiples. En primer lugar, calculo la etiqueta y el conjunto en la que debería insertarse el dato a partir mediante desplazamiento y operaciones lógicas (figura 4.2).Seguidamente compruebo si esa etiqueta se encuentra ya en el conjunto seleccionado.

En este caso la etiqueta tendrá s-d bits para poder diferenciar a cada uno de los bloques de Mp que pueden ubicarse en el mismo conjunto de Mc. El directorio cache en correspondencia asociativa por conjuntos contendrá, pues, un registro de s-d bits por cada conjuntos de líneas de Mc (el esquema lógica de una cache asociativa por conjuntos se muestra en la figura 4.3).

Si no se produce acierto y se está realizando una carga, o bien la cache es de escritura directa y tampoco se ha producido acierto, entonces se realiza el reemplazamiento de etiquetas. En cambio si se está realizando una escritura, se marca como modificada la etiqueta escogida.

Este procedimiento anterior se aplica cuando se está realizando un acceso simple, si el acceso tiene un tamaño mayor, entonces se puede dar el caso de que necesite recorrer también el bloque siguiente, es decir, los datos no tienen por qué encontrarse alineados. Esto se soluciona iterando el proced-

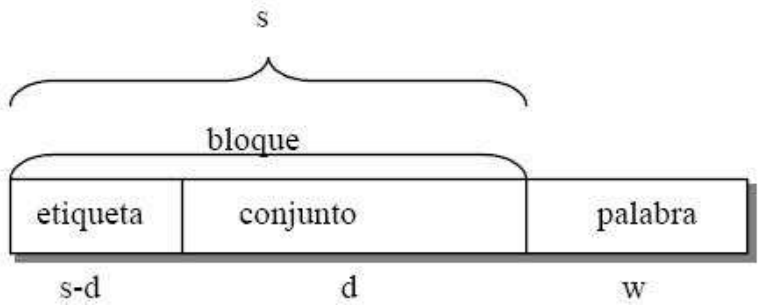

Figura 4.2: Distribución de los bits de la dirección con una correspondencia asociativa por conjuntos

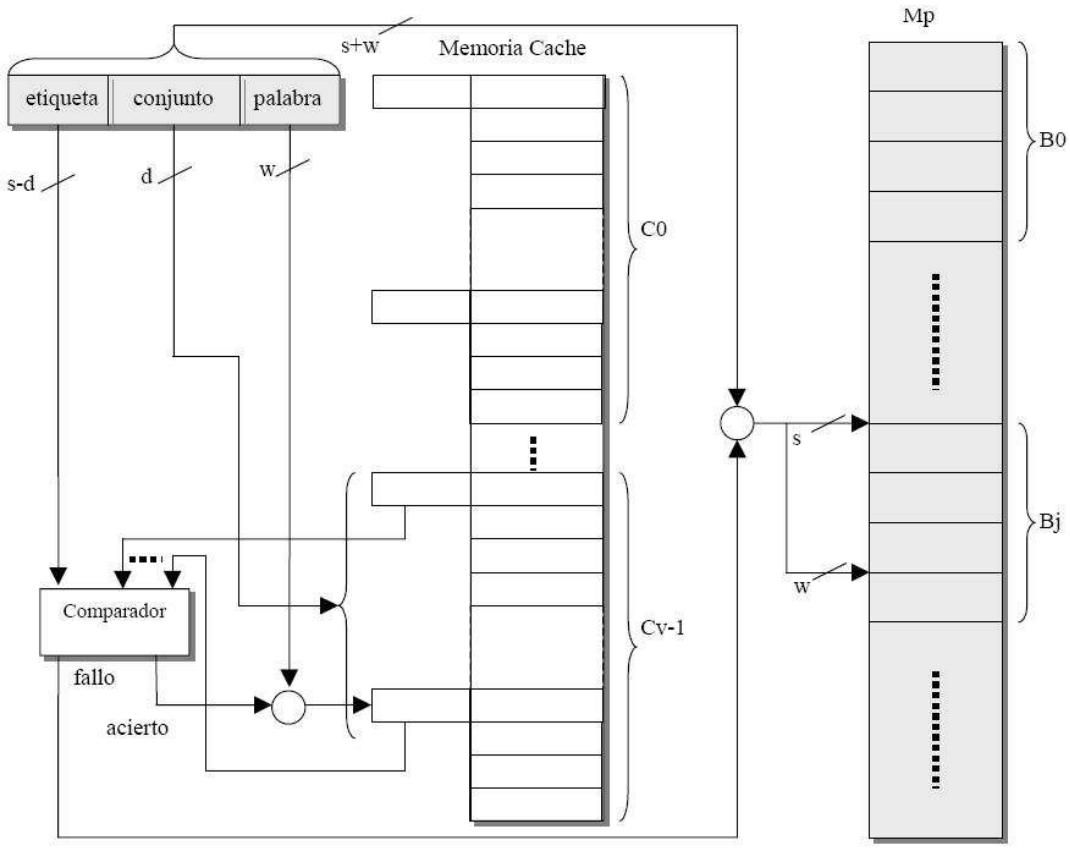

Figura 4.3: Esquema de una cache asociativa por conjuntos

imiento anterior con una dirección en la que en cada paso se va incrementando, apuntando cada vez al bloque siguiente, hasta que llego al tope fijado por el tamaño del dato más su dirección original.

4.3. Haciendo una cache adaptativa

Cada cache contiene una función que se encarga de realizar el muestreo necesario, y si es necesario, en el caso de que la tasa de aciertos actual esté por debajo de la tasa mínima o por encima de la tasa máxima, modificar el comportamiento de la cache. Esto puede hacerse seleccionando si se duplican las vías por conjunto o si se duplica el número de conjuntos (para mejorar la tasa de acierto) o dividiendo vías o conjuntos (para minimizar la utilización del hardware).

4.3.1. Duplicando vías

Si me encuentro por debajo del máximo de asociatividad fijado, multiplico tanto la asociatividad como el tamaño de la cache por 2, actualizando las máscaras correspondientes

4.3.2. Dividiendo vías

Si no he sobrepasado el límite mínimo de asociatividad fijado a priori, divido entre 2 tanto el tamaño total de la cache, como la asociatividad, actualizando las máscaras. Como se puede observar aquí no se eliminan bloques explícitamente, pero al reducir el tamaño de cada conjunto, nos deshacemos de bloques que podrían tener algún contenido válido, por lo que será necesario su volcado a memoria.

4.3.3. Duplicando conjuntos

Al duplicar conjuntos, si me encuentra por debajo del límite máximo de conjuntos, duplico el tamaño de la cache y creo nuevo espacio para los nuevos conjuntos, habiendo realizado un volcado del contenido anterior de toda la cache.

4.3.4. Dividiendo conjuntos

Si el número de conjuntos actual es distinto de 1, entonces podemos dividir los conjuntos, que consiste en reducir a la mitad el tamaño de la cache, actualizar máscaras, y realojar los conjuntos anteriores en este nuevo espacio de mitad de tamaño.

4.4. Generador de trazas en formato DIN para Dinero IV y para icache/dcache

Para validar el correcto funcionamiento de nuestras caches, se hace necesaria la utilización de algún otro simulador de caches, usaremos Dinero IV para ello. Para simular el comportamiento de una cache (con ambos simuladores) mientras se ejecuta exactamente la misma aplicación, procedemos a generar la traza de la ejecución del programa entero, para procesarla a posteriori, estáticamente.

Para ello se han implementado dos pintools, itracegen y dtracegen, que generan trazas en formato DIN con las direcciones de las instrucciones en el primer caso, y con las posiciones de memoria referenciadas a lo largo del programa mediante las cargas y los almacenamientos, en el segundo.

Posteriormente, hemos modificado las pintools que ya instrumentaban programas dinámicamente (icache y dcache), para que fueran además capaces de procesar las trazas correspondientes estáticamente, cambiando su funcionamiento mediante la línea de parámetros con la opción -e traza.

Formato DIN:

Consta de una dirección precedida de un número que indica si la dirección se refiere a una instrucción (2), una carga (0) o un almacenamiento (1): tipo dirección

Capítulo 5

Pruebas del simulador sobre varios SPEC y validación de resultados

Los SPEC CPU2000 han sido desarrollados para proporcionar un conjunto de programas de prueba heterogéneos que todos podamos usar. De este modo se pretende estandarizar los programas de prueba que se usan para evaluar los sistemas computadores, así como las mejoras que se proponen a éstos. Nos valdremos de ellos para validar los datos obtenidos con el simulador y ara probar la eficacia de las medidas adaptativas que hemos implementado. Todos los benchmarks usados pertenecen al tipo CINT2000.

En la sección 5.1 explicaremos el sistema cache propuesto. En la sección 5.2 corroboraremos los resultados obtenidos en las simulaciones comparándolos con los resultados de Dinero IV. En la siguiente sección, simularemos los mismos benchmarks pero utilizando mecanismos de adapación. El objetivo es comparar los resultados obtenidos variando los umbrales de adaptación, comprobar el efecto sobre la tasa de fallos y los posibles beneficios.

5.1. Descripción de la cache:

Se simulará una jerarquía de memoria cache formada por un nivel de datos, un nivel de instrucciones y un nivel compartido. A continuación las características de cada uno de ellos:

- Primer nivel de la cache de datos:
  - tamaño :32*KILO
  - tamaño bloque: 128
  - vías iniciales: 2

    - vías máximas: 2
    - vías mínimas: 1
    - tasa duplicar vías: 0,80
    - tasa dividir vías: 0,90

- Primer nivel de la cache de instrucciones:
    - tamaño: 64*KILO
    - tamaño bloque: 128
    - vías iniciales: 1
    - vías máximas: 1
    - vías mínimas: 1
    - tasa duplicar vías: 0,80
    - tasa dividir vías: 0,90
- Nivel compartido
    - tamaño: 1024*KILO
    - tamaño bloque: 128
    - vías iniciales: 8
    - vías máximas: 8
    - vías mínimas: 1
    - tasa duplicar vías: 0,80
    - tasa dividir vías: 0,90

5.2. Validación de resultados con Dinero IV

Para validar el simulador implementado era necesario tomar algún otro simulador como referencia. Uno de los más usados es Dinero IV, pero al ser un simulador estático era necesario implementar la posibilidad de leer trazas en formato DIN para llevar a cabo la simulación y poder comparar los resultados. En primer lugar generamos las trazas de diversos benchmarks con nuestras pintools *dtracegen* e *itracegen*, sin permitir adaptación (ya que Dinero IV no la soporta) y después simulamos.

La figura 5.1 muestra como los resultados obtenidos por ambos simuladores para el nivel 1 de datos son similares en todos los casos.

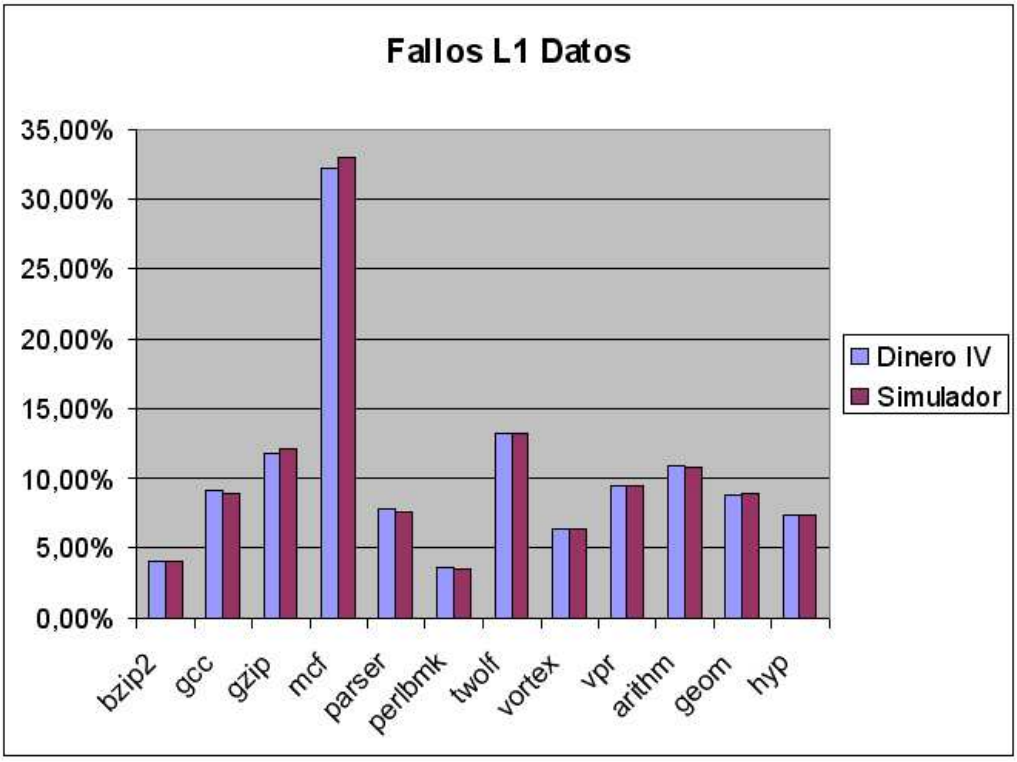

Figura 5.1: Comparativa de tasas de fallos en el nivel 1 de datos

En segundo lugar la figura 5.2 muestra los resultados para la cache de instrucciones. Se vuelve a comprobar que el margen de error entre ambos simuladores es muy reducido. Además comprobamos que la tasa de fallos es muy baja. Con una única vía el funcionamiento del nivel de instrucciones es eficiente, no es necesario en principio considerar aplicar en él mecanismos adaptativos.

Por último, la figura 5.3 representa los resultados obtenidos en el nivel compartido, vuelve a hacerse patente la similitud de los resultados obtenidos con uno y otro simulador. Consideramos que el simulador desarrollado en nuestro proyecto queda validado.

5.3. Pruebas sobre SPEC

Los SPEC utilizados en la simulación son los mismos que en la sección anterior: bzip2, gcc, gzip, mcf, parser, perlbmk, twolf, vortex, vpr, arithm, hyp y geom. Con cada uno de ellos se realizaron cinco pruebas: sin adaptación y con adaptación utilizando cuatro pares de umbrales distintos (10 %-20 %;

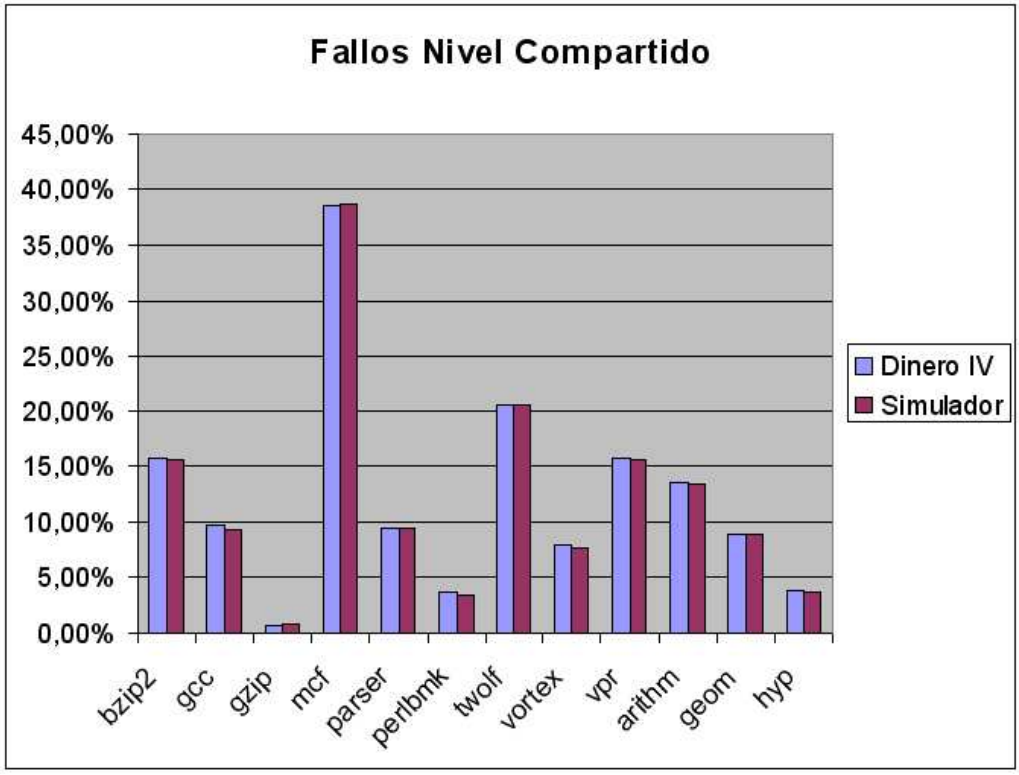

Figura 5.2: Comparativa de tasas de fallos en el nivel 2 compartido

20 %-30 %, 30 %-40 % y 40 %-50 %). En cada una de estas pruebas se estudió la tasa de fallos de la ejecución y el número medio de vías de la cache. Además, se varió el intervalo de muestreo: cada 100 mil instrucciones y cada millón de instrucciones. Las tablas de las figuras 5.4, 5.5, 5.6 y 5.7 reflejan los resultados obtenidos.

5.3.1. Conclusiones

En todas las simulaciones se comprueba que la penalización en la tasa de fallos puede considerarse despreciable, sin embargo sí puede apreciarse un ahorro energético al reducir el número medio de vías necesario.

Este ahorro es más notable en la cache compartida, donde el intervalo de vías es más amplio En función de los pares de umbrales elegidos, la tasa de fallos y el número medio de vías varía, habría que establecer un compromiso que nos lleve a una tasa de fallos mínima y al máximo ahorro de energía. Dependerá de la naturaleza de la aplicación ejecutada, en concreto de su tasa de fallos cuando no se utilizan mecanismos de adaptación, que sea óptimo

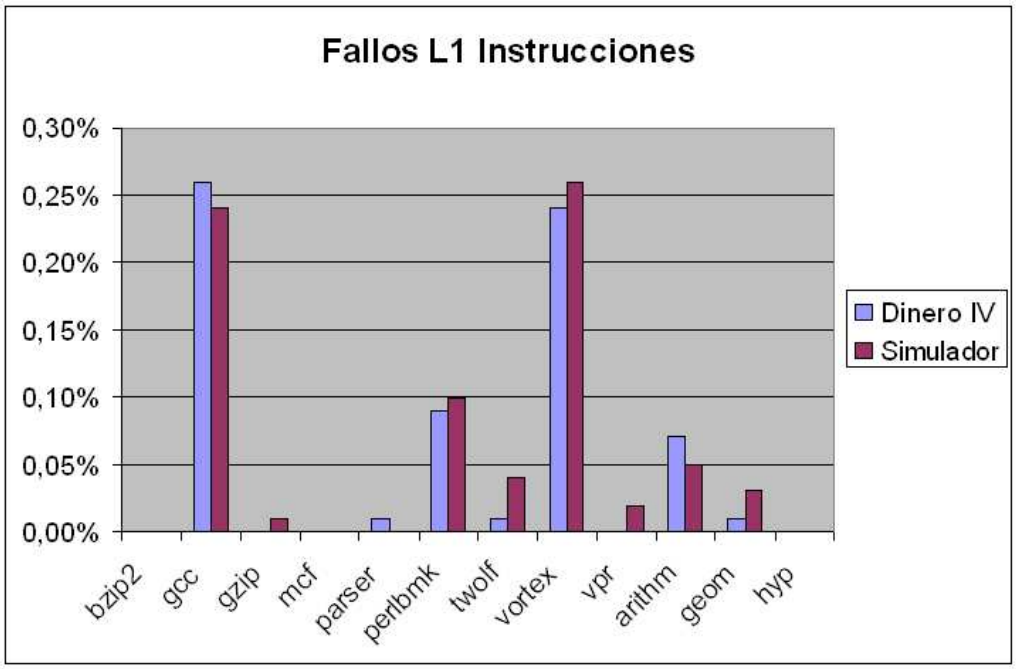

Figura 5.3: Comparativa de tasas de fallos en el nivel 1 de instrucciones

FALLOS L1 DATOS									
	Miss sin adaptar	10%-20%		20%-30%		30%-40%		40%-50%	
SPEC		Miss	Viasμ	Miss	Viasμ	Miss	Viasμ	Miss	Viasμ
bzip2	3,99%	4,40%	1,5	4,20%	1,39	4,00%	1,35	4,30%	1,34
gcc	9,00%	9,20%	1,7	9,30%	1,51	9,40%	1,46	9,30%	1,44
gzip	12,05%	12,05%	2	12,10%	1,52	12,60%	1,43	12,40%	1,41
mcf	32,95%	32,95%	2	32,95%	2	32,95%	2	33,65%	1,6
parser	7,61%	8,10%	1,8	8,20%	1,6	8,13%	1,54	8,06%	1,51
perlbmk	3,56%	3,99%	1,3	4,01%	1,21	3,77%	1,19	3,24%	1,18
twolf	13,33%	13,33%	2	13,35%	1,81	13,67%	1,65	13,44%	1,61
vortex	6,42%	6,88%	1,51	6,76%	1,39	6,72%	1,37	6,78%	1,35
vpr	9,40%	9,40%	2	9,50%	1,8	9,85%	1,76	9,87%	1,72
arithm	10,78%	10,78%	2	10,90%	1,72	10,98%	1,66	11,16%	1,62
geom	8,86%	8,88%	1,78	9,08%	1,67	9,27%	1,64	9,25%	1,62
hyp	7,40%	7,41%	1,84	7,73%	1,51	7,57%	1,42	7,82%	1,39

Figura 5.4: Tabla de resultados en el nivel L1 de datos. Muestreo cada 100 mil instrucciones

un rango de umbrales u otro.

Para establecer el par de umbrales a aplicar se podría estudiar el pro-

FALLOS L1 DATOS: muestreo cada 1000K instrucciones								
	Miss	10%-20%		20%-30%		30%-40%		40%-50%	
SPEC		Miss	Viasμ	Miss	Viasμ	Miss	Viasμ	Miss	Viasμ
bzip2	3,99%	4,35%	1,6	4,19%	1,42	4,03%	1,37	4,24%	1,38
gcc	9,00%	9,18%	1,73	9,27%	1,59	9,22%	1,5	9,29%	1,47
gzip	12,05%	12,10%	2	12,11%	1,55	12,64%	1,44	12,43%	1,44
mcf	32,95%	32,95%	2	32,95%	2	32,96%	2	33,41%	1,68
parser	7,61%	8,01%	1,81	8,07%	1,8	8,11%	1,6	8,05%	1,56
perlbmk	3,56%	3,87%	1,4	4,00%	1,23	3,76%	1,2	3,10%	1,22
twolf	13,33%	13,33%	2	13,51%	1,88	13,65%	1,69	13,82%	1,7
vortex	6,42%	6,46%	1,54	6,52%	1,44	6,63%	1,4	6,64%	1,38
vpr	9,40%	9,40%	2	9,66%	1,84	9,81%	1,76	9,60%	1,7
arithm	10,78%	10,77%	2	11,12%	1,79	11,07%	1,67	11,27%	1,72
geom	8,86%	9,22%	1,9	9,29%	1,7	9,03%	1,64	9,15%	1,7
hyp	7,40%	7,55%	1,85	7,62%	1,52	7,62%	1,47	7,87%	1,44

Figura 5.5: Tabla de resultados en el nivel L1 de datos. Muestreo cada millón de instrucciones

FALLOS L2 COMPARTIDA: muestreo cada 100K instrucciones									
	Miss	10%-20%		20%-30%		30%-40%		40%-50%	
SPEC		Miss	Viasμ	Miss	Viasμ	Miss	Viasμ	Miss	Viasμ
bzip2	15,68%	15,72%	7,80	15,77%	7,20	15,77%	7,20	15,77%	7,20
gcc	9,68%	9,72%	7,70	9,77%	6,90	9,77%	6,90	9,77%	6,90
gzip	0,66%	0,92%	5,40	0,92%	5,40	0,92%	5,40	0,92%	5,40
mcf	38,46%	38,46%	8,00	38,46%	8,00	38,46%	8,00	38,57%	7,80
parser	9,43%	9,54%	7,60	9,56%	7,57	9,56%	7,57	9,56%	7,57
perlbmk	3,59%	3,99%	6,30	3,99%	6,30	3,99%	6,30	3,99%	6,30
twolf	20,64%	20,64%	8,00	20,66%	7,89	20,72%	7,80	20,72%	7,80
vortex	7,98%	8,13%	7,50	8,13%	7,50	8,13%	7,50	8,13%	7,50
vpr	15,75%	15,75%	8,00	15,82%	7,80	15,95%	6,80	15,95%	6,80
arithm	13,54%	13,54%	8,00	13,54%	6,70	13,54%	6,70	13,54%	6,70
geom	8,87%	9,12%	6,80	9,15%	6,60	9,15%	6,60	9,15%	6,60
hyp	3,88%	4,02%	6,40	4,02%	6,40	4,02%	6,40	4,02%	6,40

Figura 5.6: Tabla de resultados en el nivel L2 compartido. Muestreo cada 100 mil instrucciones

grama previamente, por ejemplo mediante profiling, aunque para no tener que analizar el programa entero, se podría aplicar sólo a un porcentaje del código. Esta solución nos puede indicar una posible tasa de fallos orientativa y con ella elegir un rango de umbrales óptimo para la adaptación.

En cuanto al intervalo de muestreo, se aprecia que cuanto mayor sea,

FALLOS L2 COMPARTIDA: muestreo cada 1000K instrucciones									
	Miss	10%-20%		20%-30%		30%-40%		40%-50%	
SPEC		Miss	Viasμ	Miss	Viasμ	Miss	Viasμ	Miss	Viasμ
bzip2	15,68%	15,72%	7,90	15,77%	7,32	15,77%	7,30	15,77%	7,38
gcc	9,68%	9,72%	7,83	9,77%	6,98	9,77%	6,79	9,77%	6,94
gzip	0,66%	0,92%	5,70	0,92%	5,63	0,92%	5,79	0,92%	5,42
mcf	38,46%	38,46%	8,00	38,46%	8,00	38,46%	8,00	38,57%	7,88
parser	9,43%	9,54%	7,62	9,56%	7,60	9,56%	7,67	9,56%	7,70
perlbmk	3,59%	3,99%	6,36	3,99%	6,36	3,99%	6,35	3,99%	6,66
twolf	20,64%	20,64%	8,00	20,66%	7,90	20,72%	7,88	20,72%	7,82
vortex	7,98%	8,13%	7,55	8,13%	7,60	8,13%	7,57	8,13%	7,71
vpr	15,75%	15,75%	8,00	15,82%	7,81	15,95%	6,83	15,95%	6,82
arithm	13,54%	13,54%	8,00	13,54%	6,73	13,54%	6,77	13,54%	6,73
geom	8,87%	9,12%	6,91	9,15%	6,62	9,15%	6,90	9,15%	6,66
hyp	3,88%	4,02%	6,44	4,02%	6,42	4,02%	6,44	4,02%	6,70

Figura 5.7: Tabla de resultados en el nivel L2 compartido. Muestreo cada millón de instrucciones

menor será el ahorro energético. Esto se debe a que el número de posibles transformaciones de cache es menor. Sin embargo, cuanto menor sea dicho intervalo, mayor sobrecarga introduciremos en el sistema. De nuevo, habría que establecer un compromiso entre lo que queremos conseguir y el coste que estamos dispuestos a asumir.

Capítulo 6
Bibliografía

Albo00 D. Albonesi. "Selective Cache Ways: On-Demand Cache Resource Allocation". IEEE Journal on Instruction Level Parallelism, 2, 2000.

Albo98 D. Albonesi. "Dynamic IPC/Clock Rate Optimization". International Symposium on Computer Architecture, 282?292, Junio 1998.

AgLR02 v A. Agarwal, H. Li, y K. Roy. "DRG-cache: a data retention gated-ground cache for low power". Conference on Design automation, 473-478, Junio 2002.

AuLE02 T. Austin, E. Larson, y D. Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling". IEEE Computer, 35(2), Febrero 2002.

BABD00 R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, y S. Dwarkadas. "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures". International Symposium on Microarchitecture, 245-257, Diciembre 2000.

BaDA01 R. Balasubramonian, S. Dwarkadas, y D.H. Albonesi. "Reducing the Complexity of the Register File in Dynamic Superscalar Processors". 34th International Symposium on Microarchitecture, pg. 237-248, Diciembre 2001.

BaMo02 A. Baniasadi y A. Moshovos. "Asymmetric-frequency clustering: a power-aware back-end for high-performance processors". International Symposium on Low Power Electronics and Design, 255-258, Agosto 2002.

BKAB03 A. Buyuktosunoglu, T. Karkhanis, D.H. Albonesi y P. Bose. "Energy Efficient Co-Adaptive Instruction Fetch and Issue". International Symposium on Computer Architecture, Junio 2003.

BPSB00 T. Burd, T. Pering, A. Stratakos, y R. Brodersen. "A dynamic voltage scaled microprocessor system". IEEE Solid-State Circuits Conference, 294-295, 2000.

BSB+00 A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, y D.H. Albonesi. "An Adaptive Issue Queue for Reduced Power at High Performance". Workshop on Power-Aware Computer Systems, held at the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Noviembre 2000.

ChAD04 L. Chen, D.H. Albonesi, y S. Dropsho. "Dynamically Matching ILP Characteristics Via a Heterogeneous Clustered Microarchitecture". IBM Watson Conference on the Interaction Between Architecture, Circuits, and Compilers, pg. 136-143, Octubre 2004.

ChTM01 B. R. Childers, H. Tang, y R. Melhem. "Adapting Processor Supply Voltage to Instruction-Level Parallelism". Proceedings of the Kool Chips Workshop (MICRO-34), Diciembre 2001.

DBB+02 S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, y M. L. Scott. "Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power". International Conference on Parallel Architectures and Compilation Techniques, 141, Septiembre 2002.

DhSm02 A.S. Dhodapkar y J.E. Smith. "Managing Multiconfiguration Hardware via Dynamic Working Set Analysis". International Symposium on Computer Architecture, 233-244, Mayo 2002.

DKA+02 S. Dropsho, V. Kursun, D.H. Albonesi, S. Dwarkadas, y E.G. Friedman. "Managing Static Leakage Energy in Microprocessor Functional Units". 35th International Symposium on Microarchitecture, pg. 321-332, Noviembre 2002.

DSA+04 S. Dropsho, G. Semeraro, D.H. Albonesi, G. Magklis, y M.L. Scott. "Dynamically Trading Frequency for Complexity in a GALS Microprocessor". International Symposium on Microarchitecture, pg. 157-168, Diciembre 2004.

FlRM02 K. Flautner, S. Reinhardt, y T. Mudge. "Automatic performance setting for dynamic voltage scaling". Wireless Networks, 8(5), Septiembre 2002.

FoGo01 D. Folegnani y A. González. "Energy-effective issue logic". International Symposium on Computer Architecture, 230-239, Mayo 2001.

GLF+00 D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey III, y M. Neufeld. "Policies for Dynamic Clock Scheduling". 2000.

HsKH01 Chung-Hsing Hsu, Ulrich Kremer, y Michael Hsiao. "Compiler-directed dynamic voltage/frequency scheduling for energy reduction in mircoprocessors". International Symposium on Low Power Electronics and Design, 275-278, Agosto 2001.

HuRT03 M. C. Huang, J. Renau, y J. Torrellas. "Positional adaptation of processors: application to energy reduction". International Symposium on Computer Architecture, 157-168, Mayo 2003.

IyMa02 A. Iyer, y D. Marculescu. "Power and performance evaluation of globally asynchronous locally synchronous processors". International Conference on Computer Architecture. 158?168. Mayo 2002.

JWP+05 P. Juang, Q. Wu, L. S. Peh, M. Martonosi, y D. W. Clark. "Coordinated, distributed, formal energy management of chip multiprocessors". Proceedings of the 2005 International Symposium on Low Power Electronics and Design, pg. 127-130, Agosto 2005.

KaHM01 S. Kaxiras, Z. Hu, y M. Martonosi. "Cache decay: exploiting generational behavior to reduce cache leakage power". International Symposium on Computer Architecture, 29(2), Mayo 2001.

KFBM02 N. S. Kim, K. Flautner, D. Blaauw, y T. Mudge. "Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction". International Symposium on Microarchitecture, 219-230, Noviembre 2002.

KFBM04 N. S. Kim, K. Flautner, D. Blaauw, y T. Mudge. "Single-vdd and single-vt super-drowsy techniques for low-leakage high-performance instruction caches". International Symposium on Low Power Electronics and Design, Agosto, 2004.

KiGM97 J. Kin, M. Gupta, y W. H. Mangione-Smith. "The filter cache: an energy efficient memory structure". International Symposium on Microarchitecture, 184-193, Noviembre 1997.

Krew05 K. Krewell. "Yonah Does Dual-Core Right". The Insider?s Guide to Microprocessor Hardware, Marzo 2005.

Marc00 D. Marculescu. "On the Use of Microarchitecture-Driven Dynamic Voltage Scaling". Workshop on Complexity-Effective Design (ISCA-27), Junio 2000.

MSS+03 G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, y S. Dropsho. "Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor". International Symposium on Computer Architecture, Junio 2003.

MTC+03 S.W. Moore, G.S. Taylor, P.A. Cunningham, R.D. Mullins, y P. Robinson. "Self-Calibrating Clocks for Globally Asynchronous Locally Synchronous Systems". International Conference on Computer Design: VLSI in Computers & Processors (ICCD'00), p. 73-78, 2003.

PeBB00 T. Pering, T. Burd, y R. Brodersen. "Voltage scheduling in the lparm microprocessor system". International Symposium on Low Power Design, Agosto 2000.

PoKG01 D. Ponomarev, G. Kucuk, y K. Ghose. "Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources". International Symposium on Microarchitecture, 90-101, Diciembre 2001.

PoLS01 J. Pouwelse, K. Langendoen, y H. Sips. "Energy priority scheduling for variable voltage processors". International Symposium on Low Power Electronics and Design, 28-33, Agosto 2001.

PYFV01 M. Powell, S. Yang, B. Falsafi, y K. T. N. Vijaykumar. "Reducing leakage in a high-performance deep-submicron instruction cache". IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 9(1), Febrero 2001.

SaSk04 K. Sankaranarayanan y K. Skadron. "Profile-based adaptation for cache decay". ACM Transactions on Architecture and Code Optimization (TACO), 1(3), Septiembre 2004.

SMB+02 G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, y M. L. Scott. "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling". International Symposium on High-Performance Computer Architecture, Febrero 2002.

SPHC02 T. Sherwood, E. Perelman, G. Hamerly, y B. Calder. "Automatically Characterizing Large Scale Program Behavior". International Conference on Architectural Support for Programming Languages and Operating Systems, Octubre 2002.

WJM+05 Q. Wu, P. Juang, M. Martonosi, L. S. Peh, y D. W. Clark. "Formal Control Techniques for Power-Performance Management," IEEE Micro, vol. 25, num. 5, pg. 52-62, Septiembre-Octubre 2005.

XiMM03 F. Xie, M. Martonosi, y S. Malik. "Compile-time dynamic voltage scaling settings: opportunities and limits". Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pg. 49-62, Junio 2003.[XiMM03] F. Xie, M. Martonosi, y S. Malik. "Compile-time dynamic voltage scaling settings: opportunities and limits". Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pg. 49-62, Junio 2003.

XHD+02 W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, M. J. Irwin. "Compiler-directed instruction cache leakage optimization". International Symposium on Microarchitecture, Noviembre 2002.

XuAl99 B. Xu y D.H. Albonesi. "A Methodology for the Analysis of Dynamic Application Parallelism and Its Application to Reconfigurable Computing". SPIE International Conference on Reconfigurable Technology: FPGAs for Computing and Applications, 78-86, Septiembre 1999.

YPF+01 S. Yang, M. D. Powell, B. Falsafi, K. Roy, y T. N. Vijaykumar. "An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches". International Symposium on High-Performance Computer Architecture, 147-158, Enero 2001.

YPFV02 S. H. Yang, M. D. Powell, B. Falsafi, y T. N. Vijaykumar. "Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay". International Symposium on High-Performance Computer Architecture, 151, Febrero 2002.

ZaVN03 C. Zhang, F. Vahid, W. Najjar. "A highly configurable cache architecture for embedded systems". International Symposium on Computer Architecture, Junio 2003.

Apéndice A
Código comentado del simulador de caches adaptativas

A.1. cache.h

```
//
// Autores: Sergio Carazo Alba, José Carlos Silva Cuevas y Rubén Nogales Cadenas
// Este archivo contiene el código necesario que simula un nivel de cache adaptativa
//

#ifndef PIN_CACHE_H
#define PIN_CACHE_H
#define KILO 1024
#define MEGA (KILO*KILO)
#define GIGA (KILO*MEGA)

// Este tipo sirve como contador de los hit/miss de la cache
typedef UINT64 CACHE_STATS;

//Comprueba si n es potencia de 2
static inline bool IsPower2(UINT32 n) {
    return ((n & (n - 1)) == 0);
}

// Calcula log2(n) por abajo, devuelve -1 si n == 0
static inline INT32 FloorLog2(UINT32 n) {
    INT32 p = 0;
    if (n == 0) return -1;
    if (n & 0xffff0000) { p += 16; n > >= 16; }
    if (n & 0x0000ff00) { p += 8; n > >= 8; }
    if (n & 0x000000f0) { p += 4; n > >= 4; }
    if (n & 0x0000000c) { p += 2; n > >= 2; }
```

```
    if (n & 0x00000002) { p += 1; }
    return p;
}

//Calcula log2(n)por arriba, devuelve -1 si n == 0
static inline INT32 CeilLog2(UINT32 n) {
    return FloorLog2(n - 1) + 1;
}

//Tipo enumerado para el estado de un bloque en cache
typedef enum{
    Invalido,
    Valido,
    Modificado
} ESTADO_CACHE;

//Clase para los Tags de cache
class CACHE_TAG {
public:
    ADDRINT _tag;
    ADDRINT _tagi;
    ESTADO_CACHE estado;
    //Constructora
    CACHE_TAG(ADDRINT tag = 0, ADDRINT tagi = 0){
    _tag = tag; _tagi = tagi; estado = Invalido; }

    //Operador de igualdad
    bool operator==(const CACHE_TAG &right) const { return _tag == right._tag; }

    //Devuelve el valor del tag
    operator ADDRINT() const { return _tag; }
};


Clase para un conjunto de cache (tipo ROUND ROBIN), compuesto por un array de tags
namespace CACHE_SET {

class ROUND_ROBIN {
    private:
        UINT32 maxasoc;
        UINT32 minasoc;
        CACHE_TAG *_tags;
        UINT32 _tagsLastIndex;
```

```
    UINT32 _nextReplaceIndex;

public:
//Constructora
ROUND_ROBIN(UINT32 associativity ,UINT32 mxasoc, UINT32 mnasoc){
    maxasoc = mxasoc;
    minasoc = mnasoc;
    _tags = new CACHE_TAG[maxasoc];
    _tagsLastIndex=associativity - 1;
    ASSERTX(associativity >= minasoc);
    ASSERTX(associativity <= maxasoc);
    _nextReplaceIndex = _tagsLastIndex;
    UINT32 i = 0;
    for (i = 0; i <maxasoc; i++){
        _tags[i] = CACHE_TAG();
    }
}

//Accesores

UINT32 getMaxAssociativity(){return maxasoc;}
UINT32 getMinAssociativity(){return minasoc;}
CACHE_TAG getTag(UINT32 i){return _tags[i];}
UINT32 GetAssociativity(UINT32 associativity)  return _tagsLastIndex + 1;

//Vacía la cache, simula un volcado en memoria
VOID Flush(){
    for (UINT32 index = 0; index < _tagsLastIndex; index++){
        _tags[index] = CACHE_TAG(0);
    }
}

//Fija la asociatividad del conjunto
VOID SetAssociativity(UINT32 associativity){
    ASSERTX(associativity <= maxasoc);
    _tagsLastIndex = associativity - 1;
    _nextReplaceIndex = _tagsLastIndex;
}



//Algoritmo que busca un bloque en el conjunto
UINT32 Find(CACHE_TAG tag){
```

```
        bool result = true;
        for (INT32 index = _tagsLastIndex; index >= 0; index–){
            //Aunque el código queda poco elegante, supone una optimización en ensamblador
            if(_tags[index] == tag) goto end;
        }
        result = false;
        end: return result;
    }
    //Si se encuentra el bloque, se cambia su estado a modificado
    void Marcar(CACHE_TAG tag){
        bool find = false;
        for (INT32 index = _tagsLastIndex; (index >= 0) && (!find); index–){
            if(_tags[index] == tag){
            _tags[index].estado = Modificado;
            find = true;
            }
        }
    }

    //Reemplaza un bloque siguiendo la política de reemplazo Round Robin
    void Replace(CACHE_TAG tag){
        const UINT32 index = _nextReplaceIndex;
        _tags[index] = tag;
        _tags[index]._tagi = tag._tagi;
        _tags[index].estado = Valido;
        _nextReplaceIndex = (index == 0 ? _tagsLastIndex : index -1);
    }

};//class RoundRobin
};//namespace CACHE_SET



//Tipo enumerado que indica si la cache es de escritura directa o de post-escritura
namespace CACHE_ALLOC {
    typedef enum{
        STORE_ALLOCATE,
        STORE_NO_ALLOCATE
    }STORE_ALLOCATION;
};


//La siguiente clase solo implementa operaciones básicas de la cache, sin tener en cuenta
```

```
//cosas como el emplazamiento, políticas de escritura, etc.
class CACHE_BASE{
    public:
    //Tipo de acceso
    typedef enum{
        ACCESS_TYPE_LOAD,
        ACCESS_TYPE_STORE,
        ACCESS_TYPE_NUM
    }ACCESS_TYPE;

    protected:
    static const UINT32 HIT_MISS_NUM = 2;
    CACHE_STATS _access[ACCESS_TYPE_NUM][HIT_MISS_NUM];

    public:
    //Parámetros
    const std::string _name;
    UINT32 _cacheSize;
    UINT32 _lineSize;
    UINT32 _associativity;
    UINT32 _lineShift;
    UINT32 _setIndexMask;

    //Calcula el numero total de aciertos o fallos de cache en función de hit
    CACHE_STATS SumAccess(bool hit) const{
        CACHE_STATS sum = 0;
        for (UINT32 accessType = 0; accessType < ACCESS_TYPE_NUM; accessType++){
            sum += _access[accessType][hit];
        }
        return sum;
    }

    protected:
    UINT32 NumSets() const { return _setIndexMask + 1; }

    public:
    //constructoras/destructoras
    CACHE_BASE(std::string name, UINT32 cacheSize, UINT32 lineSize, UINT32 associativity);

    //accessores
    UINT32 CacheSize() const { return _cacheSize; }
    UINT32 LineSize() const { return _lineSize; }
    UINT32 Associativity() const { return _associativity; }
```

```
    //Devuelven accesos, aciertos y fallos
    CACHE_STATS Hits(ACCESS_TYPE accessType) const { return _access[accessType][true];}
    CACHE_STATS Misses(ACCESS_TYPE accessType) const { return _access[accessType][false];}
    CACHE_STATS Accesses(ACCESS_TYPE accessType) const {
        return Hits(accessType) + Misses(accessType);
    }
    CACHE_STATS Hits() const { return SumAccess(true);}
    CACHE_STATS Misses() const { return SumAccess(false);}
    CACHE_STATS Accesses() const { return Hits() + Misses();}

    //Calculan la etiqueta a partir de la dirección
    VOID SplitAddress(const ADDRINT addr, CACHE_TAG & tag, UINT32 & setIndex) const{
        tag._tag = addr > > _lineShift;
        tag._tagi = addr > > _lineShift+1;
        setIndex = tag._tag & _setIndexMask;
    }

    VOID SplitAddress(const ADDRINT addr, CACHE_TAG & tag, UINT32 & setIndex,
            UINT32 & lineIndex) const{
        const UINT32 lineMask = _lineSize - 1;
        lineIndex = addr & lineMask;
        SplitAddress(addr, tag, setIndex);
    }

    //Esta función añade a prefix los resultados obtenidos en la simulación
    string StatsLong(string prefix = ) const;
};

    //Constructora CACHE_BASE::CACHE_BASE(std::string name, UINT32 cacheSize,
            UINT32 lineSize, UINT32 associativity):
    _name(name),
    _cacheSize(cacheSize),
    _lineSize(lineSize), //2 ^w
    _associativity(associativity),
    _lineShift(FloorLog2(lineSize)) //w
    {
        //numero de conjuntos -1 , usado para calcular el tag de una dir
        _setIndexMask= (cacheSize / (associativity *lineSize))-1;
        //Si _lineSize o _setIndexMask+1 no son potencias de 2, lanzaría una excepción
        ASSERTX(IsPower2(_lineSize));
        ASSERTX(IsPower2(_setIndexMask + 1));
```

```
//Inicializa todos los tipos de accesos a cero (hits y misses)
for (UINT32 accessType = 0; accessType < ACCESS_TYPE_NUM; accessType++){
    _access[accessType][false] = 0;
    _access[accessType][true] = 0;
}   }

//Devuelve un string con las estadisticas string
CACHE_BASE::StatsLong(string prefix) const {
    const UINT32 headerWidth = 19;
    const UINT32 numberWidth = 10;
    string out;
    out += prefix + _name + ":"+ "\n";
    for (UINT32 i = 0; i ¡ACCESS_TYPE_NUM; i++){

        const ACCESS_TYPE accessType = ACCESS_TYPE(i);

        std::string type(accessType == ACCESS_TYPE_LOAD ? "Load": "Store");

        out += prefix + ljstr(type + "Hits: ", headerWidth)
        + decstr(Hits(accessType), numberWidth)
        +  +fltstr(100.0 * Hits(accessType) / Accesses(accessType), 2, 6) + " %\n";

        out += prefix + ljstr(type + "Misses: ", headerWidth)
        + decstr(Misses(accessType), numberWidth)
        +  +fltstr(100.0 * Misses(accessType) / Accesses(accessType), 2, 6) + " %\n";

        out += prefix + ljstr(type + .Accesses: ", headerWidth)
        + decstr(Accesses(accessType), numberWidth)
        +  +fltstr(100.0 * Accesses(accessType) / Accesses(accessType), 2, 6) + " %\n";

        out += prefix + "\n";
    }
    out += prefix + ljstr("Total-Hits: ", headerWidth)
    + decstr(Hits(), numberWidth)
    +  +fltstr(100.0 * Hits() / Accesses(), 2, 6) + " %\n";

    out += prefix + ljstr("Total-Misses: ", headerWidth)
    + decstr(Misses(), numberWidth)
    +  +fltstr(100.0 * Misses() / Accesses(), 2, 6) + " %\n";

    out += prefix + ljstr("Total-Accesses: ", headerWidth)
    + decstr(Accesses(), numberWidth)
    +  +fltstr(100.0 * Accesses() / Accesses(), 2, 6) + " %\n";o
```

```
        out += "\n";
        return out;
    }


//Implementacion final de una cache, hereda de cache base
using namespace CACHE_SET; class CACHE : public CACHE_BASE {
    private:
    ROUND_ROBIN * _sets;
    UINT32 max_sets;
    UINT32 store_allocation;
    public:
    //constructoras/destructoras
    CACHE(std::string name, UINT32 cacheSize, UINT32 lineSize, UINT32 associativity,
            UINT32 max_sts,UINT32 sa,UINT32 maxasoc,UINT32 minasoc)
            :CACHE_BASE(name, cacheSize, lineSize, associativity){
        max_sets=max_sts;
        store_allocation=sa;
        //El numero de conjunto ha de ser menor que el mínimo permitido
        ASSERTX(NumSets() <= max_sets);
        _sets=(ROUND_ROBIN*)malloc(sizeof(ROUND_ROBIN)*max_sets);
        for (UINT32 i = 0; i ¡NumSets(); i++){
            ROUND_ROBIN aux(associativity,maxasoc,minasoc);
            _sets[i]=aux;
        }
    }

    //Este algoritmo es el que activa o desactiva modulos de cache
    // en funcion de las tasa de aciertos
    string MuestreaNivel(float tasaAlta, float tasaBaja, int opcion){
        string out=;
        if (Accesses() !=0){
            float tasa=(float)Hits()/Accesses();
            out +=fltstr(tasa,2,6) + ;
            if (tasa> tasaAlta){
              if (opcion == 0){
                int conj=NumSets();
                if(DivideConjuntos()){
                  out+= decstr(conj,0); //num conjuntos anterior
                  out+= ;
                  out+= decstr(NumSets(),0); //num conjuntos nuevo
                  out+= "\n";
```

```
            }
            else
              out+= "\t No se puede dividir mas los conjuntos \n";
          }
          else{
            int vias=_associativity;
            if(DivideVias()){
              out +=decstr(vias,0) + ;
            out+= decstr(_associativity,0) + "\n";
              }
            else out+="\t No se puede dividir mas las vias \n";
          }
        }
        else
          if (tasa < tasaBaja) {
            if (opcion == 0){
            int conj=NumSets();
            if(DuplicaConjuntos()){
              out+=decstr(conj,0)+;
              out+= decstr(NumSets(),0)+"\n";
            }
            else out+="\t No se puede duplicar mas los conjuntos \n";
            }
            else{
              int vias=_associativity;
              if(DuplicaVias()){
              out +=decstr(vias,0) + ;
              out+= decstr(_associativity,0)+"\n";
              }
              else out+="\t No se puede duplicar mas las vias \n";
            }
          }
      }
      return out;
  }

  ROUND_ROBIN dame_set(UINT32 i){return _sets[i];}

// Si el acceso supera el tamaño del bloque, se divide el 'dato' segun ese tamaño
  bool Access(ADDRINT addr, UINT32 size, ACCESS_TYPE accessType);

//Esta llamada es para accesos que no superan el tamaño del bloque
  bool AccessSingleLine(ADDRINT addr, ACCESS_TYPE accessType);
```

```
//Funciones que transforman la cache
    bool DuplicaConjuntos();
    bool DivideConjuntos();
    bool DivideVias();
    bool DuplicaVias();

//Función auxiliar que muestra el contenido de la cache
    void MuestraCache();

};

    //Este algoritmo saca el contenido de la cache por pantalla
    VOID CACHE::MuestraCache(){
        for(UINT32 i = 0; i<NumSets(); i++){
            std:: cout < < "\t- Conjunto: "<< i << endl;
            for(UINT32 j = 0; j< _associativity; j++)
              std::cout<<"\t\t # "<< _sets[i].getTag(j)._tag <<" "<<
              _sets[i].getTag(j)._tagi<<" ´´<<_sets[i].getTag(j).estado << endl;
        }
    }

    //Algoritmo que divide por dos el número de vías de la cache siempre que sea posible
    bool CACHE::DivideVias(){
        if(_associativity/2 >= _sets[0].getMinAssociativity()){
            _cacheSize /= 2;
            _associativity /=2;
            _setIndexMask = (_cacheSize / (_associativity * _lineSize)) -1;
            for(UINT32 i = NumSets(); i<NumSets(); i++){
              _sets[i].SetAssociativity(_associativity);
            }
            return true;
        }
        else return false;
    }

    //Algoritmo que divide el número de vías de la cache por dos, siempre que sea posible
    bool CACHE::DuplicaVias(){
        if(_associativity*2 <= _sets[0].getMaxAssociativity()){
            _cacheSize *= 2;
            _associativity *=2;
            _setIndexMask = (_cacheSize / (_associativity * _lineSize))-1;
            for(UINT32 i = NumSets(); i<NumSets(); i++){
```

```
            _sets[i].SetAssociativity(_associativity);
        }
        return true;
    }
    else return false;
}
//Algoritmo que divide el número de conjuntos de la cache entre dos
bool CACHE::DivideConjuntos(){
    if(NumSets()/2>=1){
        _cacheSize /= 2;
        _setIndexMask = (_cacheSize / (_associativity * _lineSize)) - 1;
        _sets = (ROUND_ROBIN*)realloc(_sets,NumSets()*sizeof(ROUND_ROBIN));
        for(UINT32 i = NumSets()/2; i<NumSets(); i++){
            _sets[i].SetAssociativity(_associativity);
        }
        return true;
    }
    else return false;
}

//Algoritmo que duplica el numero de conjuntos de la cache
bool CACHE::DuplicaConjuntos(){
    if(NumSets()*2<=max_sets){
        for(UINT32 j = 0; j¡NumSets();j++){
            _sets[j].Flush();
        }
        _cacheSize *= 2;
        _setIndexMask = (_cacheSize / (_associativity * _lineSize)) - 1;
        _sets = (ROUND_ROBIN*)realloc(_sets,NumSets()*sizeof(ROUND_ROBIN));
        for(UINT32 i = NumSets()/2; i<NumSets(); i++){
            _sets[i].SetAssociativity(_associativity);
            _sets[i].Flush();
        }
        return true;
    }
    else return false;
}

//Acceso a la cache, el dato al que se accede excede el tamaño del bloque
bool CACHE::Access(ADDRINT addr, UINT32 size, ACCESS_TYPE accessType){
    ADDRINT highAddr = addr + size;
    //ultima direccion del 'dato'
    bool allHit = true;
```

```
    //Si es cierto, todos los bloques del 'dato' están en cache
    ADDRINT lineSize = LineSize();
    ADDRINT notLineMask =  (lineSize - 1);
    do{
        CACHE_TAG tag;
        UINT32 setIndex;
        //Saca el tag y obtenemos el conjunto
        SplitAddress(addr, tag, setIndex);
        //set es un puntero de tipo SET, es el conjunto donde estamos buscando
        ROUND_ROBIN & set = _sets[setIndex];
        //si esta el tag en el set, localHit es true
        bool localHit = set.Find(tag);
        //en el momento en el que falle un bloque, allHit sera false
        allHit &= localHit;
        //En caso de fallo, si es store y cache tipo allocate, o simplemente si es un load,
        //se almacena el bloque
        if(((!localHit) && (accessType ==ACCESS_TYPE_LOAD ||
        store_allocation ==CACHE_ALLOC::STORE_ALLOCATE))){
          set.Replace(tag);
        }
        if (accessType == ACCESS_TYPE_STORE)
          set.Marcar(tag);
        //Continuamos con el siguiente bloque
        addr = (addr & notLineMask) + lineSize;
    }while (addr < highAddr);
    _access[accessType][allHit]++;
    return allHit;
}

//Acceso a la cache, el dato al que se accede no supera el tamaño de bloque
bool CACHE::AccessSingleLine(ADDRINT addr, ACCESS_TYPE accessType){
    CACHE_TAG tag;
    UINT32 setIndex;
    SplitAddress(addr, tag, setIndex);
    ROUND_ROBIN & set = _sets[setIndex];
    bool hit =set.Find(tag);
        //En caso de fallo, si es store y cache tipo allocate, o simplemente si es un load,
        //se almacena el bloque
        if(((! hit) && (accessType == ACCESS_TYPE_LOAD ||
        store_allocation == CACHE_ALLOC::STORE_ALLOCATE))){
          set.Replace(tag);
        }
        if (accessType == ACCESS_TYPE_STORE)
          set.Marcar(tag);
```

```
            _access[accessType][hit]++;
            return hit;
    }

#endif //PIN_CACHE_H
```

A.2. sysmem.h

```
//
// Autores: Sergio Carazo Alba, José Carlos Silva Cuevas y Rubén Nogales Cadenas
// Este archivo contiene el código necesario para extender a varios niveles de cache, del mismo tipo
//

#include "cache.h"
#include <iostream>
#include <fstream>

class SysMem {

private:
    CACHE** niveles ;
    UINT32 numNiveles;
    string nombre;
    float tasaAlta;
    float tasaBaja;
    int opcion;

public:
    //Constructora, los parámetros se leen de fichero, así no es necesario recompilar
    SysMem(char* fichero,string name){
        nombre=name;
        const int sz=100;
        char buffer [sz] ;
        ifstream in(fichero);
        do{
            strcpy(buffer,);
            in.getline(buffer,sz);
            sscanf(buffer," %f%f%d",&tasaAlta,&tasaBaja,&opcion);
        }while(buffer[0]=='#');
        int niv;
```

```
do{
    strcpy(buffer,);
    in.getline(buffer,sz);
    sscanf(buffer," %d",&niv);
}while(buffer[0]=='#');

numNiveles=niv;
niveles = new CACHE*[numNiveles];
UINT32 tamC,lS,asoc,alloc,maxasoc,minasoc;

for(UINT32 i=0;i<numNiveles;i++){
do{
    strcpy(buffer,);
    in.getline(buffer,sz);
    sscanf(buffer," %d *KILO", &tamC);
}while (buffer[0]=='#');
do{
    strcpy(buffer,);
    in.getline(buffer,sz);
    sscanf(buffer," %d ", &lS);
}while (buffer[0]=='#');
do{
    strcpy(buffer,);
    in.getline(buffer,sz);
    sscanf(buffer," %d ", &asoc);
}while (buffer[0]=='#');
do{
    strcpy(buffer,);
    in.getline(buffer,sz);
    sscanf(buffer," %d ", &alloc);
}while (buffer[0]=='#');
do{
    strcpy(buffer,);
    in.getline(buffer,sz);
    sscanf(buffer," %d ", &maxasoc);
}while (buffer[0]=='#');
do{
    strcpy(buffer,);
    in.getline(buffer,sz);
    sscanf(buffer," %d ", &minasoc);
}while (buffer[0]=='#');

UINT32 maxSets = tamC*KILO / (lS * asoc);
```

```
    niveles[i]=(CACHE*)malloc(8*sizeof(CACHE));
    char name1[10]=;
    sprintf(name1,"L %d",i);
    niveles[i] = new CACHE(name1,tamC*KILO,lS,asoc,maxSets,alloc,maxasoc,minasoc);
    }
}

UINT32 dameNiveles(){return numNiveles;}

//Muestra los parámetros de la cache. leídos de fichero
void mostrarParametros(){
    for (UINT32 i = 0; i<numNiveles; i++){
        cout << "Nivel :"<< i << endl;
        cout << niveles[i]->_cacheSize << endl;
        cout << niveles[i]->_lineSize << endl;
        cout << niveles[i]->_associativity << endl;
    }
}

//Esta funcion es llamada periódicamente con el fin de "adaptar"
//cada nivel de la cache según la tasa de aciertos actual
string muestrea(string str){
    string out=str;
    string aux=" ";
    string aux2=" ";
    for(UINT32 i=0; i<numNiveles;i++){
        aux + = nombre + " :\n # Nivel: " + decstr(i+1,0);
        aux += "\n";
        aux2= niveles[i]->MuestreaNivel(tasaAlta,tasaBaja,opcion);
        aux+=aux2;
        out +=aux;
    }
    return out;
}

//Acceso a un dato cuyo tamaño es menor que un bloque
bool accesoSimple(ADDRINT addr, CACHE_BASE::ACCESS_TYPE accessType){
    bool hit= false;
    UINT32 i=1;
    hit = niveles[0]->AccessSingleLine(addr,accessType);
    while(!hit && i < numNiveles){
        hit =niveles[i]->AccessSingleLine(addr,CACHE_BASE::ACCESS_TYPE_LOAD);
        i++;
```

```
    };
    return hit;
}

//Acceso a un dato cuyo tamaño es mayor que un bloque
bool accesoMultiple(ADDRINT addr, UINT32 size,
        CACHE_BASE::ACCESS_TYPE accessType){
    bool hit=false;
    UINT32 i=1;
    hit = niveles[0]-¿Access(addr,size,accessType);
    while(!hit && i < numNiveles) {
        hit = niveles[i]-¿Access(addr,size,CACHE_BASE::ACCESS_TYPE_LOAD);
        i++;
    };
    return hit;
}

//Muestra el contenido de cada nivel de cache
void MuestraCaches(){
    for(UINT32 i = 0; i<numNiveles;i++){
        std::cout<<"Nivel :"<< i<<endl;
        this-¿niveles[i]-¿MuestraCache();
    }
}

//Muestra los resultados de cada nivel de cache
void imprimeResultados(const char *file){
    std::ofstream out(file);
    for(UINT32 i=0;i<numNiveles;i++){
        // Imprime los resultados
        out << "#Nivel: "<<i<<endl;

        out << "\t# Numero de accesos totales: "
        <<decstr(niveles[i]->Accesses(),0) << " accesos\n\n";

        out << "\t# Numero de aciertos: "
        << decstr(niveles[i]->Hits(),0) << "aciertos\n";

        out << "\t# Tasa de aciertos: "
        << fltstr(100.0 * niveles[i]->Hits() / niveles[i]->Accesses(), 2, 6) << "%\n\n";

        out << "\t# Numero de fallos: "
        << decstr(niveles[i]->Misses(),0) << "fallos\n";
```

```
            out << "\t# Tasa de fallos:"
            << fltstr(100.0 * niveles[i]->Misses() / niveles[i]->Accesses(), 2, 6) << "%\n\n";

            out << "#"<< endl;
        }

        out.close();
    }

    //Añade a str las stadísitcas obtenidas en la simulación
    string statsLong(string str){
        string out = str;
        string aux=;
        for(UINT32 i=0; i<numNiveles;i++){
            aux = niveles[i]->StatsLong();
            out+=aux;
        }
        return out;
    }
};
```

A.3. idsysmem.h

```
//
// Autores: Sergio Carazo Alba, José Carlos Silva Cuevas y Rubén Nogales Cadenas
// Este archivo contiene el código necesario para extender a un sistema cache de
// instrucciones y datos, con un nivel compartido
//

#include "sysmem.h"
#include <iostream>
#include <fstream>

class IdSysMem{

private:
    SysMem* Datos;
    SysMem* Inst;
    SysMem* Comp;
```

```
public:
    //Constructora, los parámetros se leen de archivos
    IdSysMem(char* ficheroDatos, char* ficheroInst, char* ficheroComp){
        Datos = new SysMem(ficheroDatos, "Datos");
        Inst = new SysMem(ficheroInst,"Instrucciones");
        Comp = new SysMem(ficheroComp,Çompartida");
    }

    //Muestra los parámetros leídos de cada nivel
    void mostrarParametros(){
        std::cout << "Caches Datos" << endl;
        Datos->mostrarParametros();
        std::cout << "Caches Instrucciones" << endl;
        Inst->mostrarParametros();
        std::cout << "Caches Compartidas" << endl;
        Comp->mostrarParametros();
    }

    //Muestrea cada nivel del sistema para activar o desactivar
    //modulos en caso de que sea necesario
    string muestrea(string str){
        string out1,out2,out3;
        out1 = Datos->muestrea(str);
        out2 = Inst->muestrea(out1);
        out3 = Comp->muestrea(out2);
        return out3;
}

    //Se accede a un dato cuyo tamaño es menor que el de bloque
    void accesoSimpleDatos(ADDRINT addr, CACHE_BASE::ACCESS_TYPE accessType){
        bool hit= false;
        if (Datos->dameNiveles()!= 0){
            hit = Datos->accesoSimple(addr,accessType);
            if((!hit) && (Comp->dameNiveles()!=0))
              hit = Comp->accesoSimple(addr,CACHE_BASE::ACCESS_TYPE_LOAD);
        }
    }



    //Se accede a un dato cuyo tamaño es mayor que el de bloque
    void accesoMultipleDatos(ADDRINT addr, UINT32 size,
```

```
        CACHE_BASE::ACCESS_TYPE accessType){
    bool hit=false;
    if(Datos->dameNiveles()!= 0){
        hit = Datos-¿accesoMultiple(addr,size,accessType);
        if((!hit) && (Comp->dameNiveles()!=0))
            hit = Comp->accesoMultiple(addr,size,CACHE_BASE::ACCESS_TYPE_LOAD);
    }
}

//Se accede a una instrucción cuyo tamaño es menor que el de bloque
void accesoSimpleInst(ADDRINT addr){
    bool hit= false;
    if(Inst->dameNiveles()!=0){
        hit = Inst->accesoSimple(addr,CACHE_BASE::ACCESS_TYPE_LOAD);
        if((!hit) && (Comp->dameNiveles()!=0))
            hit = Comp->accesoSimple(addr,CACHE_BASE::ACCESS_TYPE_LOAD);
    }
}

//Se accede a una instrucción cuyo tamaño es mayor que el de bloque
void accesoMultipleInst(ADDRINT addr, UINT32 size){
    bool hit=false;
    if(Inst->dameNiveles()!=0){
        hit =Inst->accesoMultiple(addr,size,CACHE_BASE::ACCESS_TYPE_LOAD);
        if((!hit) && (Comp->dameNiveles()!=0))
            hit =Comp->accesoMultiple(addr,size,CACHE_BASE::ACCESS_TYPE_LOAD);
    }
}

//Muestra el contenido de cada sistema cache, el de datos y el de instrucciones
void MuestraCaches(){
    if(Datos->dameNiveles() !=0){
        std::cout<<"Cache Datos´´ << endl;
        Datos->MuestraCaches();
        if(Comp->dameNiveles() != 0)
          Comp->MuestraCaches();
    }
    if(Inst->dameNiveles() !=0){
        std::cout<<"Cache Instrucciones"<< endl;
        Inst->MuestraCaches();
        if(Comp->dameNiveles() !=0)
          Comp->MuestraCaches();
    }
```

```
    }

    //Añade a str los resultados de la simulación
    string statsLong(string str){
        string out1 = " ";
        string out2 = " ";
        string out3 = " ";
        string aux1 = " ";
        string aux2 = " ";

        if(Comp->dameNiveles()!=0)
            out2 = Comp-¿statsLong("");

        if (Datos->dameNiveles()!=0){
            out1 ="******Cache Datos*******\n";
            out1 += Datos->statsLong(str);
            aux1 = out1 + out2;
        }

        if (Inst->dameNiveles() !=0){
            out3 ="******Cache Instrucciones******\n";
            out3 += Inst->statsLong("");
            aux2 = out3 + out2;
        }
        return (str + aux1+ aux2);
    }
};
```

A.4. dcache.c

```
//
// Autores: Sergio Carazo Alba, José Carlos Silva Cuevas y Rubén Nogales Cadenas
// Este archivo contiene la PINTOOL que simula el comportamiento de un sistema
// de memoria cache de datos
//

#include "pin.H"
#include <iostream>
#include <fstream>
#include "sysmem.h"
#include "pin_profile.H"
```

```
//Opciones de la línea de comandos


KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, “pintool”,
    “o”, “dcache.out”, “para especificar el archivo de salida”);
KNOB<UINT32> KnobPeriodo(KNOB_MODE_WRITEONCE, “pintool”,
    “p”, “1000”, “para especificar el período de muestreo”);
KNOB¡string¿KnobEjecucion(KNOB_MODE_WRITEONCE, ”pintool”,
    “e”, “programa”, “para especificar el tipo de ejecucion que
    queremos, si un programa o una traza”);



    INT32 Usage() {
        cerr << “Simulador de cache de datos.\n”
            “\n´´;
        cerr << endl;

        return -1;
    }


// Variables globales


// Contador para poder muestrear la cache periódicamente
UINT32 cuenta =0;
// La salida de la simulación
std::ofstream out;
// multilevel es el modelo de sistema de cache en sí
SysMem multilevel(“entrada.in”,“Cache datos”);

//Para ir almacenando los miss y hits que vayan apareciendo en la simulación
typedef enum {
    COUNTER_MISS = 0,
    COUNT = 1,
    COUNTER_NUM
} COUNTER;

typedef COUNTER_ARRAY<UINT64, COUNTER_NUM> COUNTER_HIT_MISS;
```

```
COMPRESSOR_COUNTER<ADDRINT, UINT32, COUNTER_HIT_MISS> profile;


//Las siguientes funciones son las usadas en la instrumentacion,
//es decir, son las que se insertan en el código original

//Acceso tipo Load cuando se excede el tamaño de bloque
void LoadMultiple(ADDRINT addr, UINT32 size, UINT32 instId){
    multilevel.accesoMultiple(addr,size,CACHE_BASE::ACCESS_TYPE_LOAD);
}

//Acceso tipo Load cuando no se excede el tamaño de bloque
void LoadSimple(ADDRINT addr, UINT32 instId){
    multilevel.accesoSimple(addr,CACHE_BASE::ACCESS_TYPE_LOAD);
}

//Acceso tipo Store cuando se excede el tamaño de bloque
void StoreMultiple(ADDRINT addr, UINT32 size, UINT32 instId) {
    multilevel.accesoMultiple(addr,size,CACHE_BASE::ACCESS_TYPE_STORE);
}

//Acceso tipo Store cuando no se excede el tamaño de bloque
void StoreSimple(ADDRINT addr, UINT32 instId){
    multilevel.accesoSimple(addr,CACHE_BASE::ACCESS_TYPE_STORE);
}

//Función de muestreo
void muestrea(){
    if(cuenta == KnobPeriodo.Value()){
        cuenta = 0;
        out << multilevel.muestrea("");
    }
}

//Esta función es la que lleva a cabo la instrumentacion en sí
VOID Instruction(INS ins, void * v){
    if (INS_IsMemoryRead(ins)){
        const UINT32 size = INS_MemoryReadSize(ins);
        const BOOL single = (size <= 4);
        if( single ){
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR) LoadSimple,
                IARG_MEMORYREAD_EA,
```

```
                IARG_END);
        }
        else{
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR) LoadMultiple,
                IARG_MEMORYREAD_EA,
                IARG_MEMORYREAD_SIZE,
                IARG_END);
        }
    }
    if( INS_IsMemoryWrite(ins)){
            const UINT32 size = INS_MemoryWriteSize(ins);
            const BOOL single = (size <= 4);
            if( single ){
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR) StoreSimple,
                IARG_MEMORYWRITE_EA,
                IARG_END);
            }
            else{
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR) StoreMultiple,
                IARG_MEMORYWRITE_EA,
                IARG_MEMORYWRITE_SIZE,
                IARG_END);
            }
    }
    cuenta = cuenta+1;
}



//Función que finaliza la simulación
VOID Fini(int code, VOID * v){
    out << "\n";
    out << "PIN:MEMLATENCIES 1.0. 0x0\n";
    out <<
        "#\n"
        "# DCACHE stats\n"
        "#\n";
    out << multilevel.statsLong("#\n ");
    out.close();
}
```

```
//Función main int main(int argc, char *argv[]){
    PIN_InitSymbols();
    if( PIN_Init(argc,argv) ){
        return Usage();
    }
    out.open(KnobOutputFile.Value().c_str());
    out << "#TRANSFORMACIONES DE CACHE\n";
    out<<"modo " <<KnobEjecucion.Value() <<"\n";
    if(KnobEjecucion.Value()=="programa"){

//Modo de simulación normal
        std::cout<<"Comienza la simulacion"<<endl;
        INS_AddInstrumentFunction(Instruction, 0);
        PIN_AddFiniFunction(Fini, 0);
        PIN_StartProgram();
        return 0;
    }
    else{

//Modo de simulacion que lee trazas en formato din
        std::cout<<"Modo lectura de traza´´"<<endl;
        const int sz=100;
        char buffer[sz];
        long dire;
        int size;
        int tipo;
        ifstream in("trazadirecta.din");
        while(in.getline(buffer,sz)){
            sscanf(buffer,"%d %lx %d",&tipo, &dire,&size);
            printf("%lx",dire);
            strcpy(buffer,"");
            const UINT32 instId = profile.Map(dire);
            if (tipo==0){
                if(size <= 4)
                    LoadSimple((ADDRINT)dire,instId);
                else
                    LoadMultiple((ADDRINT)dire,size,instId);
            }
            else if (tipo==1 ){
                if(size <= 4)
                    StoreSimple((ADDRINT)dire,instId);
```

```
                else
                    StoreMultiple((ADDRINT)dire,size,instId);
            }
        }
        multilevel.imprimeResultados("salida.out");
        return 0;
    }
}

//eof
```

A.5. icache.c

```
//
// Autores: Sergio Carazo Alba, José Carlos Silva Cuevas y Rubén Nogales Cadenas
// Este archivo contiene la PINTOOL que simula el comportamiento de un sistema
// de memoria cache de instrucciones
//

#include "pin.H"
#include <iostream>
#include <fstream>
#include "sysmem.h"
#include "pin_profile.H"


//Opciones de la línea de comandos


KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "dcache.out", "para especificar el archivo de salida");
KNOB<UINT32> KnobPeriodo(KNOB_MODE_WRITEONCE, "pintool",
    "p", "1000", "para especificar el período de muestreo");
KNOB¡string¿KnobEjecucion(KNOB_MODE_WRITEONCE, "pintool",
    "e", "programa", "para especificar el tipo de ejecucion que
    queremos, si un programa o una traza");



    INT32 Usage() {
        cerr << "Simulador de cache de instrucciones.\n"
```

```
            "\n´´;
        cerr << endl;

        return -1;
    }


// Variables globales


// Contador para poder muestrear la cache periódicamente
UINT32 cuenta =0;
// La salida de la simulación
std::ofstream out;
// multilevel es el modelo de sistema de cache en sí
SysMem multilevel("entrada.in","Cache datos");

//Para ir almacenando los miss y hits que vayan apareciendo en la simulación
typedef enum {
    COUNTER_MISS = 0,
    COUNT = 1,
    COUNTER_NUM
} COUNTER;

typedef COUNTER_ARRAY<UINT64, COUNTER_NUM> COUNTER_HIT_MISS;

COMPRESSOR_COUNTER<ADDRINT, UINT32, COUNTER_HIT_MISS> profile;


//Las siguientes funciones son las usadas en la instrumentacion,
//es decir, son las que se insertan en el código original

//Acceso cuando se excede el tamaño de bloque
void AccesoMultiple(ADDRINT addr, UINT32 size, UINT32 instId){
    multilevel.accesoMultiple(addr,size,CACHE_BASE::ACCESS_TYPE_LOAD);
}

//Acceso cuando no se excede el tamaño de bloque
void AccesoSimple(ADDRINT addr, UINT32 instId){
    multilevel.accesoSimple(addr,CACHE_BASE::ACCESS_TYPE_LOAD);
}

//Función de muestreo
```

```
void muestrea(){
    if(cuenta == KnobPeriodo.Value()){
        cuenta = 0;
        out << multilevel.muestrea("");
    }
    cuenta = cuenta+1;
}

//Esta función es la que lleva a cabo la instrumentacion en sí
VOID Instruction(INS ins, void * v){
    const ADDRINT iaddr = INS_Address(ins);
    const UINT32 instId = profile.Map(iaddr);
    const UINT32 size = INS_MemoryReadSize(ins);
    const BOOL single = (size <= 4);
    if( single ){
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, (AFUNPTR) AccesoSimple,
            IARG_ADDRINT, iaddr,
            IARG_UINT32, instId,
            IARG_END);
    }
    else{
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, (AFUNPTR) AccesoMultiple,
            IARG_ADDRINT, iaddr,
            IARG_UINT32, (int) size,
            IARG_UINT32, instId,
            IARG_END);
    }
    INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, (AFUNPTR) muestrea,
            IARG_ADDRINT, iaddr,
            IARG_END);
}


//Función que finaliza la simulación
VOID Fini(int code, VOID * v){
    out << "\n";
    out << "PIN:MEMLATENCIES 1.0. 0x0\n";
    out <<
        "#\n"
        "# ICACHE stats\n"
```

```
        "#\n";
    out << multilevel.statsLong("#\n ");
    out.close();
}


//Función main int main(int argc, char *argv[]){
    PIN_InitSymbols();
    if( PIN_Init(argc,argv) ){
        return Usage();
    }
    out.open(KnobOutputFile.Value().c_str());
    out << "#TRANSFORMACIONES DE CACHE\n";
    out<<"modo " <<KnobEjecucion.Value() <<"\n";
    if(KnobEjecucion.Value()=="programa"){

//Modo de simulación normal
        std::cout<<"Comienza la simulacion"<<endl;
        INS_AddInstrumentFunction(Instruction, 0);
        PIN_AddFiniFunction(Fini, 0);
        PIN_StartProgram();
        return 0;
    }
    else{

//Modo de simulacion que lee trazas en formato din
        const int sz=100;
        char buffer[sz];
        int contador = 0;
        long dire;
        int size;

        ifstream in("trazadirecta.din");
        while(in.getline(buffer,sz)){
            sscanf(buffer,"%d %lx %d",&tipo, &dire,&size);
            strcpy(buffer,"");
            const UINT32 instId = profile.Map(dire);
            if(size <= 4)
                AccesoSimple((ADDRINT)dire,instId);
            else
                AccesoMultiple((ADDRINT)dire,size,instId);
            if (contador==KnobPeriodo.Value()){
                out <<multilevel.muestrea("");
```

```
                contador = 0;
            }
            contador++;
        }
        out << "PIN:MEMLATENCIES 1.0. 0x0\n";
        out <<
            "#\n"
            "# ICACHE stats\n"
            "#\n";
        out << multilevel.statsLong("#\n ");
        out.close();
        return 0;
    }
}

//eof
```

A.6. idcache.c

```
//
// Autores: Sergio Carazo Alba, José Carlos Silva Cuevas y Rubén Nogales Cadenas
// Este archivo contiene la PINTOOL que simula el comportamiento de un sistema
// de memoria cache de datos y de instrucciones
//

#include "pin.H"
#include <iostream>
#include <fstream>
#include <unistd.h>
#include "idSysmem.h"
#include "pin_profile.H"


//Opciones de la línea de comandos


KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "dcache.out", "para especificar el archivo de salida");
KNOB<UINT32> KnobPeriodo(KNOB_MODE_WRITEONCE, "pintool",
    "p", "1000", "para especificar el período de muestreo");
```

```
    INT32 Usage() {
        cerr << "Simulador de cache de datos e instrucciones.\n"
            "\n´´;
        cerr << endl;

        return -1;
    }


// Variables globales


// Contador para poder muestrear la cache periódicamente
UINT32 cuenta =0;
// La salida de la simulación
std::ofstream out;
// Contador de numero de instrucciones del programa original ejecutadas
UINT64 numInstrucciones = 0;
// multilevel es el modelo de sistema de cache en sí
IdSysMem multilevel("entradaDatos.in","entradaInst.in","entradaComp.in");

//Para ir almacenando los miss y hits que vayan apareciendo en la simulación
typedef enum {
    COUNTER_MISS = 0,
    COUNT = 1,
    COUNTER_NUM
} COUNTER;

typedef COUNTER_ARRAY<UINT64, COUNTER_NUM> COUNTER_HIT_MISS;

COMPRESSOR_COUNTER<ADDRINT, UINT32, COUNTER_HIT_MISS> profile;


//Las siguientes funciones son las usadas en la instrumentacion,
//es decir, son las que se insertan en el código original

//Acceso tipo Load cuando se excede el tamaño de bloque
void LoadMultipleDatos(ADDRINT addr, UINT32 size, UINT32 instId){
    multilevel.accesoMultipleDatos(addr,size,CACHE_BASE::ACCESS_TYPE_LOAD);
}

//Acceso tipo Load cuando no se excede el tamaño de bloque
```

```
void LoadSimpleDatos(ADDRINT addr, UINT32 instId){
    multilevel.accesoSimpleDatos(addr,CACHE_BASE::ACCESS_TYPE_LOAD);
}

//Acceso tipo Store cuando se excede el tamaño de bloque
void StoreMultipleDatos(ADDRINT addr, UINT32 size, UINT32 instId) {
    multilevel.accesoMultipleDatos(addr,size,CACHE_BASE::ACCESS_TYPE_STORE);
}

//Acceso tipo Store cuando no se excede el tamaño de bloque
void StoreSimpleDatos(ADDRINT addr, UINT32 instId){
    multilevel.accesoSimpleDatos(addr,CACHE_BASE::ACCESS_TYPE_STORE);
}

//Acceso a una instruccion cuando ésta excede el tamaño de bloque
void AccesoMultipleInst(ADDRINT addr, UINT32 size, UINT32 instId) {
    multilevel.accesoMultipleInst(addr,size);
}

//Acceso a una instruccion cuando ésta no excede el tamaño de bloque
void AccesoSimpleInst(ADDRINT addr, UINT32 instId){
    multilevel.accesoSimpleInst(addr);
}

//—textitFuncion de muestreo
void muestreo(){
    if(cuenta == KnobPeriodo.Value()){
        cuenta = 0;
        out << "\n/*******Nº ins: " << decstr(numInstrucciones,0) <<"*******/\n";
        out << multilevel.muestrea("");
    }
    cuenta = cuenta+1;
}

//Esta función es la que lleva a cabo la instrumentacion en sí
VOID Instruction(INS ins, void * v){
    const ADDRINT iaddr = INS_Address(ins);
    const UINT32 instId = profile.Map(iaddr);
    const UINT32 size = INS_MemoryReadSize(ins);
    const BOOL single = (size <= 4);
    numInstrucciones++;

    //Acceso de instrucción
```

```
if( single ){
    INS_InsertPredicatedCall(
        ins, IPOINT_BEFORE, (AFUNPTR) AccesoSimpleInst,
        IARG_ADDRINT, iaddr,
        IARG_UINT32, instId,
        IARG_END);
}
else{
    INS_InsertPredicatedCall(
        ins, IPOINT_BEFORE, (AFUNPTR) AccesoMultipleInst,
        IARG_ADDRINT, iaddr,
        IARG_UINT32, (int) size,
        IARG_UINT32, instId,
        IARG_END);
}

//Acceso a datos
if (INS_IsMemoryRead(ins)){
    const UINT32 size = INS_MemoryReadSize(ins);
    const BOOL single = (size <= 4);
    if( single ){
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, (AFUNPTR) LoadSimpleDatos,
            IARG_MEMORYREAD_EA,
            IARG_END);
    }
    else{
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, (AFUNPTR) LoadMultipleDatos,
            IARG_MEMORYREAD_EA,
            IARG_MEMORYREAD_SIZE,
            IARG_END);
    }
}
if( INS_IsMemoryWrite(ins)){
        const UINT32 size = INS_MemoryWriteSize(ins);
        const BOOL single = (size <= 4);
        if( single ){
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, (AFUNPTR) StoreSimpleDatos,
            IARG_MEMORYWRITE_EA,
            IARG_END);
        }
```

```
            else{
            INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR) StoreMultipleDatos,
                IARG_MEMORYWRITE_EA,
                IARG_MEMORYWRITE_SIZE,
                IARG_END);
            }
    }
    INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, (AFUNPTR) muestreo,
            IARG_END);
}


//Función que finaliza la simulación
VOID Fini(int code, VOID * v){
    time_t tiempo;
    out << "\n";
    out << "PIN:MEMLATENCIES 1.0. 0x0\n";
    out <<
        "#\n"
        "# IDCACHE stats\n"
        "#\n";
    out << multilevel.statsLong("#\n ");
    time(&tiempo);
    out << "Fin: " << ctime(&tiempo);
    out.close();
}


//Función main int main(int argc, char *argv[]){
    time_t tiempo;
    time(&tiempo);
    PIN_InitSymbols();
    if( PIN_Init(argc,argv) ){
        return Usage();
    }
    out.open(KnobOutputFile.Value().c_str());
    out <<KnobOutputFile.Value().c_str() << "\n"
    out <<KnobPeriodo.Value() << "\n";
    out << "#TRANSFORMACIONES DE CACHE\n";
    out<<"Inicio " << ctime(&tiempo) <<"\n";
```

```
        INS_AddInstrumentFunction(Instruction, 0);
        PIN_AddFiniFunction(Fini, 0);
        PIN_StartProgram();
        return 0;
}

//eof
```

A.7. dtracegen.c

```
//
// Autores: Sergio Carazo Alba, José Carlos Silva Cuevas y Rubén Nogales Cadenas
// Este archivo contiene la PINTOOL que genera una traza en formato din de accesos a datos
// que puede leer el programa DINERO y nuestro simulador dcache
//

#include "pin.H"
#include <stdio.h>
#include <fstream>

FILE * trace;

//Las funciones que se irán insertando en el código original, tras cada instrucción
VOID printip(VOID *ip) { fprintf(trace, "2 %p\n",ip); }
VOID printlec(VOID *ip) { fprintf(trace, "0 %p\n",ip); }
VOID printesc(VOID *ip) { fprintf(trace, "1 %p\n",ip); }

//Pin llama a esta función tras cada instrucción del programa original
VOID Instruction(INS ins, VOID *v) {
    if (INS_IsMemoryRead(ins)){
        INS_InsertCall(
            ins, IPOINT_BEFORE, (AFUNPTR)printlec,
            IARG_MEMORYREAD_EA, IARG_END);
    }
    else if ( INS_IsMemoryWrite(ins) ){
        INS_InsertCall(
            ins, IPOINT_BEFORE, (AFUNPTR)printesc,
            IARG_MEMORYWRITE_EA, IARG_END);
    }
    else{
        INS_InsertCall(
```

```
            ins, IPOINT_BEFORE, (AFUNPTR)printip,
            IARG_INST_PTR, IARG_END);
    }
}

// Se llama a esta función cuando la aplicación finaliza
VOID Fini(INT32 code, VOID *v){
    fclose(trace);
}

//main de la Pintool
int main(int argc, char * argv[]) {
    trace = fopen("dtracegen.din", "w");
    // Inicializamos PIN
    PIN_Init(argc, argv);
    //Decimos a PIN que llame a la función Instruction cada vez que se ejecute una instrucción
    INS_AddInstrumentFunction(Instruction, 0);
    //Decimos a PIN que llame a la función Fini cuando finalice la aplicación
    PIN_AddFiniFunction(Fini, 0);
    //Comienza el programa
PIN_StartProgram();
return 0;
}
```

A.8. itracegen.c

```
//
// Autores: Sergio Carazo Alba, José Carlos Silva Cuevas y Rubén Nogales Cadenas
// Este archivo contiene la PINTOOL que genera una traza en formato din de instrucciones
// que puede leer el programa DINERO y nuestro simulador icache
//

#include "pin.H"
#include <stdio.h>
#include <fstream>

FILE * trace;

//Las funciones que se irán insertando en el código original, tras cada instrucción
VOID printip(VOID *ip) { fprintf(trace, "2 %p\n",ip); }
```

```
//Pin llama a esta función tras cada instrucción del programa original
VOID Instruction(INS ins, VOID *v) {
        INS_InsertCall(
            ins, IPOINT_BEFORE, (AFUNPTR)printip,
            IARG_INST_PTR, IARG_END);
}

// Se llama a esta función cuando la aplicación finaliza
VOID Fini(INT32 code, VOID *v){
    fclose(trace);
}

//main de la Pintool
int main(int argc, char * argv[]) {
    trace = fopen("itracegen.din", "w");
    // Inicializamos PIN
    PIN_Init(argc, argv);
    //Decimos a PIN que llame a la función Instruction cada vez que se ejecute una instrucción
    INS_AddInstrumentFunction(Instruction, 0);
    //Decimos a PIN que llame a la función Fini cuando finalice la aplicación
    PIN_AddFiniFunction(Fini, 0);
    //Comienza el programa
PIN_StartProgram();
return 0;
}
```

Apéndice B
Manual de referencia del API de PIN

B.1. Controlando e Inicializando

Tipos Definidos

- typedef INT32 LEVEL_PINCLIENT::THREADID
- typedef VOID(*LEVEL_PINCLIENT::REMOVE_INSTRUMENTATION_CALLBACK)(VOID *v)
- typedef VOID(*LEVEL_PINCLIENT::DETACH_CALLBACK)(VOID *v)
- typedef VOID (* LEVEL_PINCLIENT::FINI_CALLBACK)(INT32 code, VOID *v)
- typedef VOID(* LEVEL_PINCLIENT::FORK_CALLBACK)(INT32 threadid, VOID *v)
- typedef VOID(*LEVEL_PINCLIENT::THREAD_BEGIN_CALLBACK)(UINT32 threadIndex, VOID *sp, int flags, VOID *v)
- typedef VOID(*LEVEL_PINCLIENT::THREAD_END_CALLBACK)(UINT32 threadIndex, INT32 code, VOID *v)

Funciones

- VOID LEVEL_PINCLIENT::PIN_RemoveAllProbes()
- VOID LEVEL_PINCLIENT::PIN_InsertProbe (ADDRINT src_addr,ADDRINT dst_addr)
- VOID LEVEL_PINCLIENT::PIN_RemoveProbe (ADDRINT address)
- VOID LEVEL_PINCLIENT::PIN_LockClient()
- VOID LEVEL_PINCLIENT::PIN_UnlockClient ()
- VOID LEVEL_PINCLIENT::PIN_AddFiniFunction (FINI_CALLBACK fun,VOID *val)
- VOID LEVEL_PINCLIENT::PIN_AddDetachFunction (DETACH_CALLBACK fun, VOID *val)
- VOID LEVEL_PINCLIENT::PIN_AddRemoveInstrumentationFunction (REMOVE_INSTRUMENTATION_CALLBACK fun, VOID *val)

- VOID LEVEL_PINCLIENT::PIN_AddThreadBeginFunction (THREAD_BEGIN_CALLBACK fun, VOID *val)
- VOID LEVEL_PINCLIENT::PIN_AddThreadEndFunction (THREAD_END_CALLBACK fun, VOID *val)
- VOID LEVEL_PINCLIENT::PIN_RemoveInstrumentation ()
- VOID LEVEL_PINCLIENT::PIN_RemoveFiniFunctions ()
- VOID LEVEL_PINCLIENT::PIN_Detach ()
- VOID LEVEL_PINCLIENT::PIN_RawMmap (VOID *start,size_t length, int prot, int flags, int fd, off_t offset)
- VOID LEVEL_PINCLIENT::PIN_StartProgram ()
- BOOL LEVEL_PINCLIENT::PIN_Init (INT32 argc, CHAR **argv)
- VOID LEVEL_PINCLIENT::GetVmLock ()
- VOID LEVEL_PINCLIENT::ReleaseVmLock ()
- VOID LEVEL_PINCLIENT::PIN_InitSymbols ()

Descripcion Detallada

Grupo de funciones usadas para inicializar Pin, comenzar la aplicación, y funciones de retorno para eventos tales como la finalización de la aplicación (exit).

Documentación de los Tipos Definidos

- typedef VOID(* LEVEL_PINCLIENT::DETACH_CALLBACK)(VOID *v)
  Función de retorno llamada cuando Pin abandona la aplicación.
- typedef VOID(* LEVEL_PINCLIENT::FINI_CALLBACK)(INT32 code, VOID *v)
  Función de retorno llamada cuando la aplicación termina
- typedef VOID(* LEVEL_PINCLIENT::FORK_CALLBACK)(INT32 threadid, VOID *v)
  Función de retorno para el proceso hijo de una operación de tipo fork
- typedef VOID(* LEVEL_PINCLIENT::REMOVE_INSTRUMENTATION_CALLBACK)(VOID *v)
  Función de retorno llamada cuando Pin borra todo el código de instrumentación de su cache.
- typedef VOID(* LEVEL_PINCLIENT::THREAD_BEGIN_CALLBACK)(UINT32 threadIndex, VOID * sp, int flags, VOID *v)
  Función de retorno llamada cuando comienza el hilo (thread)
- typedef VOID(* LEVEL_PINCLIENT::THREAD_END_CALLBACK)(UINT32 threadIndex, INT32 code, VOID *v)
  Función de retorno llamada cuando finaliza el hilo
- typedef INT32 LEVEL_PINCLIENT::THREADID
  Todavía no está implementada. Documentación de los Tipos Definidos
- typedef VOID(* LEVEL_PINCLIENT::DETACH_CALLBACK)(VOID *v)
  Función de retorno llamada cuando Pin abandona la aplicación.
- typedef VOID(* LEVEL_PINCLIENT::FINI_CALLBACK)(INT32 code, VOID *v)
  Función de retorno llamada cuando la aplicación termina

- typedef VOID(* LEVEL_PINCLIENT::FORK_CALLBACK)(INT32 threadid, VOID *v)
  Función de retorno para el proceso hijo de una operación de tipo fork

- typedef VOID(* LEVEL_PINCLIENT::REMOVE_INSTRUMENTATION_CALLBACK)(VOID *v)
  Función de retorno llamada cuando Pin borra todo el código de instrumentación de su cache.

- typedef VOID(* LEVEL_PINCLIENT::THREAD_BEGIN_CALLBACK)(UINT32 threadIndex, VOID * sp, int flags, VOID *v)
  Función de retorno llamada cuando comienza el hilo (thread)

- typedef VOID(* LEVEL_PINCLIENT::THREAD_END_CALLBACK)(UINT32 threadIndex, INT32 code, VOID *v)
  Función de retorno llamada cuando finaliza el hilo

- typedef INT32 LEVEL_PINCLIENT::THREADID
  Todavía no está implementada.

Documentación de Funciones

- *VOID GetVmLock()*
  (Sólo para expertos.)

- *VOID PIN_AddDetachFunction(DETACH_CALLBACK fun, VOID *val)*
  Llama a una función justo después de que Pin abandone el control de la aplicación mediante PIN_Detach. La función no puede ser una función de instrumentación
  Nota: Podemos añadir varias funciones de esta manera.
  Parámetros:

  fun Llamada a la función a ejecutarse después de Pin Detach.

  val Valor que se le pasa a la función fun cuando la llamamos.

- *VOID PIN_AddFiniFunction(FINI_CALLBACK fun, VOID *val)*
  Llama a una función justo antes de que la aplicación termine. La función no puede ser de instrumentación.

  Nota: Podemos añadir varias funciones de esta manera.

  Parámetros:

  fun Llamada a la función; se le pasa val .

  val Valor que se le pasa a fun cuando es llamada.

- *VOID PIN_AddRemoveInstrumentationFunction(REMOVE_INSTRUMENTA-TION_CALLBACK fun, VOID *val)*
  Llama a una función justo después de que Pin haya eliminado todo el código de instrumentación previo de la cache y antes de empezar a aplicar cualquier nueva instrumentación del código.

  Nota: Podemos añadir varias funciones de esta manera.

  Parámetros:

  fun Llamada a la función antes mencionada.

  val Valor que se le pasa a fun cuando es llamada.

- *VOID PIN_AddThreadBeginFunction(THREAD_BEGIN_CALLBACK fun, VOID *val)*
  Llama a una función justo después de que un nuevo hilo (thread) haya sido creado.

  Parámetros:

  fun Llamada a la función antes mencionada.

  val Valor que se le pasa a fun cuando es llamada.

- *VOID PIN_AddThreadEndFunction(THREAD_END_CALLBACK fun, VOID * val)*
  Llama a una función justo antes de que finalice un hilo (thread).

  Parámetros:

  fun Llamada a la función antes mencionada.

  val Valor que se le pasa a la función fun cuando es llamada.

- *VOID PIN_Detach()*
  Pin abandona el control de la aplicación y se ejecuta el código original (sin las funciones de instrumentación). Para asegurar un comportamiento correcto de la función, ésta ha de ser llamada mediante una rutina de análisis. Si la llamamos a través de una rutina de instrumentación, Pin podría comportarse de forma inadecuada. Ver Test/detach.c

- *BOOL PIN_Init(INT32 argc, CHAR ** argv)*
  Inicializa Pin. Debe llamarse después de PIN_StartProgram

  Parámetros:

  argc Argc del main

  argv Argv del main

  Devuelve:

  "True" si se detecta un error al pasar la línea de comandos

- *VOID PIN_InitSymbols()*
  Inicializa el código de la tabla de símbolos. Pin no leerá los símbolos si no llamamos primero a esta función. Se debe llamar después de PIN_StartProgram

- *VOID PIN_InsertProbe(ADDRINT src_addr, ADDRINT dst_addr)*
  Inserta una sonda en la dirección especificada.

  Parámetros:

  src_addr Dirección en la que queremos insertar la sonda

  dst_addr Destino al que la sonda debe apuntar.

- *VOID PIN_LockClient()*
  Algunas funciones de la API de Pin deben ser llamadas cuando el hilo (thread) tiene este candado. Esta función lo coge. Ver también PIN_UnlockClient.

- *VOID PIN_RawMmap(VOID *start, size_t length, int prot, int flags, int fd, off_t offset)*

  Empieza la ejecución del programa. Primero hay que llamar a PIN_Init

- *VOID PIN_RemoveAllProbes()*
  Borra todas las sondas previamente insertadas.

- *VOID PIN_RemoveFiniFunctions()*
  Invalida todas las "funciones Fini" insertadas mediante PIN_AddFiniFunction; Estas funciones no serán llamadas mientras dure la ejecución de la aplicación.

- *VOID PIN_RemoveInstrumentation()*
  Borra toda la instrumentación. Cuando se ejecute el código de la aplicación, las rutinas de instrumentación serán llamadas para reinstrumentar" todo el código.

- *VOID PIN_RemoveProbe(ADDRINT address)*
  Borra la sonda que fue previamente insertada en la dirección especificada por .ªddress".

  Parámetros:

  address Dirección en la que fue insertada la sonda

- *VOID PIN_StartProgram()*
  Empieza la ejecución del programa. Debemos llamar antes a PIN_Init.

- *VOID PIN_UnlockClient()*
  Algunas funciones de la API de Pin deben ser llamadas cuando el hilo (thread) tiene este candado. Esta función lo libera. Ver también PIN_LockClient

- *VOID ReleaseVmLock()*
  (Sólo para expertos)

B.2. IMG: Objeto Image

Tipos Definidos

- typedef VOID(*LEVEL_PINCLIENT::IMAGECALLBACK)(IMG, VOID *)

Enumeraciones

enum LEVEL_CORE :: IMG_TYPE {IMG_TYPE_INVALID, LEVEL_CORE :: IMG_TYPE_STATIC, LEVEL_CORE :: IMG_TYPE_SHARED, LEVEL_CORE :: IMG_TYPE_SHAREDLIB, LEVEL_CORE :: IMG_TYPE_RELOCATABLE, IMG_TYPE_LAST}

Funciones

- IMG LEVEL_PINCLIENT::IMG_Next (IMG x)
- IMG LEVEL_PINCLIENT::IMG_Prev (IMG x)
- IMG LEVEL_PINCLIENT::IMG_Invalid ()
- BOOL LEVEL_PINCLIENT::IMG_Valid (IMG x)
- SEC LEVEL_PINCLIENT::IMG_SecHead (IMG x)
- SEC LEVEL_PINCLIENT::IMG_SecTail (IMG x)
- SYM LEVEL_PINCLIENT::IMG_RegsymHead (IMG x)
- ADDRINT LEVEL_PINCLIENT::IMG_Entry (IMG x)
- const string & LEVEL_PINCLIENT::IMG_Name (IMG x)
- ADDRINT LEVEL_PINCLIENT::IMG_Gp (IMG x)
- ADDRINT LEVEL_PINCLIENT::IMG_LoadOffset (IMG x)
- ADDRINT LEVEL_PINCLIENT::IMG_LowAddress (IMG x)
- ADDRINT LEVEL_PINCLIENT::IMG_HighAddress (IMG x)
- ADDRINT LEVEL_PINCLIENT::IMG_StartAddress (IMG x)
- USIZE LEVEL_PINCLIENT::IMG_SizeMapped (IMG x)
- IMG_TYPE LEVEL_PINCLIENT::IMG_Type (IMG x)
- VOID LEVEL_PINCLIENT::IMG_AddInstrumentFunction (IMAGECALLBACK fun, VOID *v)
- VOID LEVEL_PINCLIENT::IMG_AddUnloadFunction (IMAGECALLBACK fun, VOID *v)

- IMG LEVEL_PINCLIENT::IMG_Open (const string &filename)
- IMG LEVEL_PINCLIENT::APP_ImgHead ()
- IMG LEVEL_PINCLIENT::APP_ImgTail ()

Descripción detallada

IMG representa todas las estructuras de datos que le corresponden a un ejecutable (o a una librería compartida). Se puede acceder a él durante la instrumentación y también en la fase de análisis. Los objetos IMG se crean de forma perezosa. Para que un IMG se cree es necesario que se cargue una librería compartida; por tanto, en toda la vida de un proceso se puede crear multitud de objetos IMG.

Uso en la iteración:

// Recorre todas las imágenes cargadas

for(IMG img= APP_ImgHead(); IMG_Valid(img); img =IMG_Next(img))

Documentación de los tipos definidos

- typedef VOID(* LEVEL_PINCLIENT::IMAGECALLBACK)(IMG, VOID *)

Tipo de la función que se llamará cuando se cargue la imagen

Documentación de los tipos enumerados

enum LEVEL_CORE::IMG_TYPE
Valores de la enumeración:

- IMG_TYPE_STATIC
  Imagen principal, enlazado con static.
- IMG_TYPE_SHARED
  Imagen principal, enlazado con las librerías compartidas.
- IMG_TYPE_SHAREDLIB
  Librería compartida.
- IMG_TYPE_RELOCATABLE
  Objeto relocalizable (fichero).

Documentación de las funciones

- IMG APP_ImgHead()
  Devuelve:

  La primera imagen cargada en memoria
- *IMG APP_ImgTail()*
  Devuelve:

  La última imagen cargada en memoria

- *VOID IMG_AddInstrumentFunction(IMAGECALLBACK fun, VOID *v)*
  Use este procedimiento para registrar un retorno, para capturar la carga de una imagen

  Parámetros:

  fun Función de instrumentación para imágenes, se le pasa una imagen y v

  v El valor a pasar a fun cuando la imagen se carga

- *VOID IMG_AddUnloadFunction(IMAGECALLBACK fun, VOID *v)*
  Registra fun como la función a llamar cuando la imagen se descarga. Ésta no es una función de instrumentación (no tiene sentido instrumentar una función cuando se la ha eliminado de memoria).

  Parámetros:

  fun Se le pasa una imagen y v cuando la imagen se descarga

  v El valor a pasar a fun cuando la imagen se descarga

- *ADDRINT IMG_Entry(IMG x)*
  No implementada todavía.

  Devuelve:

  Dirección de la primera instrucción ejecutada cuando se carga la imagen

- *ADDRINT IMG_Gp(IMG x)*
  No implementada todavía.

  Devuelve:

  Puntero Global (Global pointer (GP)) de la imagen, si se utiliza un GO para direccionar los datos globales

- *ADDRINT IMG_HighAddress(IMG x)*
  Devuelve:

  La dirección virtual más alta de la imagen en la memoria

- *IMG IMG_Invalid()*
  Devuelve:

  Para indicar que no es imagen

- *ADDRINT IMG_LoadOffset(IMG x)*
  Devuelve:

  Desplazamiento entre la dirección cargada y la dirección enlazada de la imagen

- *ADDRINT IMG_LowAddress(IMG x)*
  Devuelve:

  La dirección virtual más baja de la imagen en la memoria

- *const string& IMG_Name(IMG x)*
  Devuelve:

  Nombre de la imagen

- *IMG IMG_Next(IMG x)*
  Devuelve:

  De la lista de las imágenes cargadas en memoria devuelve la imagen cargada después de x, o IMG_Invalid() si x es la última imagen

- *IMG IMG_Open(const string & filename)*

- *IMG IMG_Prev(IMG x)*
  Devuelve:

  De la lista de las imágenes cargadas en memoria devuelve la imagen cargada antes de x, o IMG_Invalid() si x es la primera imagen

- *SYM IMG_RegsymHead(IMG x)*
  Devuelve:

  Primer símbolo regular en la imagen

- *SEC IMG_SecHead(IMG x)*
  Devuelve:

  Primera sección de la imagen

- *SEC IMG_SecTail(IMG x)*
  Devuelve:

  Última sección de la imagen

- *USIZE IMG_SizeMapped(IMG x)*
  Devuelve:

  Tamaño de la imagen

- *ADDRINT IMG_StartAddress(IMG x)*
  Devuelve:

  La dirección virtual de comienzo de la imagen

- *IMG_TYPE IMG_Type(IMG x)*
  Devuelve:

  Tipo de imagen

- *BOOL IMG_Valid(IMG x)*
  Devuelve:

  True si x no es IMG_Invalid()

SEC: Objeto Section

Enumeraciones enum LEVEL_CORE::SEC_TYPE { SEC_TYPE_INVALID, SEC_TYPE_UNUSED, LEVEL_CORE::SEC_TYPE_REGREL, LEVEL_CORE::SEC_TYPE_DYNREL, LEVEL_CORE::SEC_TYPE_EXEC, LEVEL_CORE::SEC_TYPE_DATA, SEC_TYPE_DYNAMIC, SEC_TYPE_OPD, SEC_TYPE_GOT, SEC_TYPE_STACK, SEC_TYPE_PLTOFF, SEC_TYPE_HASH, LEVEL_CORE::SEC_TYPE_LSDA, SEC_TYPE_UNWIND, SEC_TYPE_UNWINDINFO, SEC_TYPE_REGSYM, SEC_TYPE_DYNSYM, SEC_TYPE_DEBUG, LEVEL_CORE::SEC_TYPE_BSS, SEC_TYPE_SYMSTR, SEC_TYPE_DYNSTR, SEC_TYPE_SECSTR, SEC_TYPE_COMMENT, SEC_TYPE_LAST }

Funciones

- IMG LEVEL_PINCLIENT::SEC_Img (SEC x)
- SEC LEVEL_PINCLIENT::SEC_Next (SEC x)
- SEC LEVEL_PINCLIENT::SEC_Prev (SEC x)
- SEC LEVEL_PINCLIENT::SEC_Invalid ()
- BOOL LEVEL_PINCLIENT::SEC_Valid (SEC x)
- RTN LEVEL_PINCLIENT::SEC_RtnHead (SEC x)

- RTN LEVEL_PINCLIENT::SEC_RtnTail (SEC x)
- const string & LEVEL_PINCLIENT::SEC_Name (SEC x)
- SEC_TYPE LEVEL_PINCLIENT::SEC_Type (SEC x)
- BOOL LEVEL_PINCLIENT::SEC_Mapped (SEC sec)
- const VOID * LEVEL_PINCLIENT::SEC_Data (SEC x)
- ADDRINT LEVEL_PINCLIENT::SEC_Address (SEC sec)
- BOOL LEVEL_PINCLIENT::SEC_IsReadable (SEC sec)
- BOOL LEVEL_PINCLIENT::SEC_IsWriteable (SEC sec)
- BOOL LEVEL_PINCLIENT::SEC_IsExecutable (SEC sec)
- USIZE LEVEL_PINCLIENT::SEC_Size (SEC sec)

Descripcion detallada

PIN se encarga de instrumentar porciones de código introduciendo instrucciones propias cuyos resultados pueden ser analizados posteriormente. Para llevar a cabo esto, los programas que hacen uso de PIN pueden dividir el código. Un SEC es una sección del código de instrucciones, que a su vez puede ser dividido en rutinas que a su vez pueden ser descompuestas en instrucciones. Un SEC se modela tras encontrar la sección correspondiente dentro de la imagen . Se puede acceder a él durante la instrumentación y durante el análisis.

Los SECs pueden estar cargados en memoria o no, un SEC cargado en memoria ocupa espacio de direcciones dentro de la imagen IMG. Un SECs no cargado en memoria suele contener debugs o información de relocalización.

Como hacer las iteraciones:

```
// Para pasar por cada una de las secciones de una imagen
for( SEC sec= IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec) )

// Para pasar por cada una de las secciones de una imagen en sentido inverso
for( SEC sec= IMG_SecTail(img); SEC_Valid(sec); sec = SEC_Prev(sec) )
```

Documentación de los tipos de enumeración

enum LEVEL_CORE::SEC_TYPE

Valores de la enumeración:

- SEC_TYPE_REGREL relocalizaciones
- SEC_TYPE_DYNREL relocalizaciones dinámicas
- SEC_TYPE_EXEC contiene el código

- SEC_TYPE_DATA contiene los datos inicializados
- SEC_TYPE_LSDA Información de Excepciones antigua (está obsoleta)
- SEC_TYPE_BSS contiene datos no inicializados

Documentación de las funciones

- *ADDRINT SEC_Address(SEC sec)*
  Devuelve:

  La dirección de memoria de sec
- *ADDRINT SEC_Address(SEC sec)*
  Devuelve:

  La dirección de memoria de sec
- *const VOID* SEC_Data(SEC x)*
  Devuelve:

  Para un sec no cargado en memoria, devuelve un puntero a los datos del sec
- *IMG SEC_Img(SEC x)*
  Devuelve:

  La imagen que contiene a la sección
- *SEC SEC_Invalid()*
  Devuelve:

  Un valor no válido como sección
- *BOOL SEC_IsExecutable(SEC sec)*
  Devuelve:

  TRUE si la sección se puede ejecutar
- *BOOL SEC_IsReadable(SEC sec)*
  Devuelve:

  TRUE si la sección se puede leer
- *BOOL SEC_IsWriteable(SEC sec)*
  Devuelve:

  TRUE si se puede escribir en la sección (editar)
- *BOOL SEC_Mapped(SEC sec)*
  Devuelve:

  True si la sección esta cargada en memoria. Una sección no cargada contiene datos que no son necesarios en tiempo de ejecución, como información de debug
- *const string& SEC_Name(SEC x)*
  Devuelve:

  El nombre de la sección
- *SEC SEC_Next(SEC x)*
  Devuelve:

  La sección que sigue a x, o SEC_Invalid() si x es la última sección de la imagen

- *SEC SEC_Prev(SEC x)*
  Devuelve:

  La sección anterior a x, o SEC_Invalid() si x es la primera de la imagen

- *RTN SEC_RtnHead(SEC x)*
  Devuelve:

  La primera rutina de x o RTN_Invalid() si no tiene rutinas

- *RTN SEC_RtnTail(SEC x)*
  Devuelve:

  La última rutina de x, o RTN_Invalid() si no tiene rutinas

- *USIZE SEC_Size(SEC sec)*
  Devuelve:

  El tamaño de la sección

- *SEC_TYPE SEC_Type(SEC x)*
  Devuelve:

  El tipo de la sección

- *BOOL SEC_Valid(SEC x)*
  Devuelve:

  True si x es una sección válida, esto es, no es una

- *SEC_Invalid()const VOID* SEC_Data(SEC x)*
  Devuelve:

  Para un sec no cargado en memoria, devuelve un puntero a los datos del sec

- *IMG SEC_Img(SEC x)*
  Devuelve:

  La imagen que contiene a la sección

- *SEC SEC_Invalid()*
  Devuelve:

  Un valor no válido como sección

- *BOOL SEC_IsExecutable(SEC sec)*
  Devuelve:

  TRUE si la sección se puede ejecutar

- *BOOL SEC_IsReadable(SEC sec)*
  Devuelve:

  TRUE si la sección se puede leer

- *BOOL SEC_IsWriteable(SEC sec)*
  Devuelve:

  TRUE si se puede escribir en la sección (editar)

- *BOOL SEC_Mapped(SEC sec)*
  Devuelve:

  True si la sección esta cargada en memoria. Una sección no cargada contiene datos que no son necesarios en tiempo de ejecución, como información de debug

- *const string& SEC_Name(SEC x)*
  Devuelve:

  El nombre de la sección

- *SEC SEC_Next(SEC x)*
  Devuelve:

  La sección que sigue a x, o SEC_Invalid() si x es la última sección de la imagen

- *SEC SEC_Prev(SEC x)*
  Devuelve:

  La sección anterior a x, o SEC_Invalid() si x es la primera de la imagen

- *RTN SEC_RtnHead(SEC x)*
  Devuelve:

  La primera rutina de x o RTN_Invalid() si no tiene rutinas

- *RTN SEC_RtnTail(SEC x)*
  Devuelve:

  La última rutina de x, o RTN_Invalid() si no tiene rutinas

- *USIZE SEC_Size(SEC sec)*
  Devuelve:

  El tamaño de la sección

- *SEC_TYPE SEC_Type(SEC x)*
  Devuelve:

  El tipo de la sección

- *BOOL SEC_Valid(SEC x)*
  Devuelve:

  True si x es una sección válida, esto es, no es una SEC_Invalid()

B.3. RTN: Objeto Routine

Tipos Definidos

typedef VOID(*LEVEL_PINCLIENT::RTN_INSTRUMENT_CALLBACK)(RTN rtn, VOID *v)

Funciones

- SEC LEVEL_PINCLIENT::RTN_Sec (RTN x)
- RTN LEVEL_PINCLIENT::RTN_Next (RTN x)
- RTN LEVEL_PINCLIENT::RTN_Prev (RTN x)
- RTN LEVEL_PINCLIENT::RTN_Invalid ()
- BOOL LEVEL_PINCLIENT::RTN_Valid (RTN x)
- BBL LEVEL_PINCLIENT::RTN_BblHead (RTN x)

- BBL LEVEL_PINCLIENT::RTN_BblTail (RTN x)
- const string & LEVEL_PINCLIENT::RTN_Name (RTN x)
- AFUNPTR LEVEL_PINCLIENT::RTN_ConvertToFunptr (RTN x)
- INT32 LEVEL_PINCLIENT::RTN_No (RTN x)
- VOID LEVEL_PINCLIENT::RTN_AddInstrumentFunction (RTN_INSTRUMENT_CALLBACK fun, VOID *val)
- USIZE LEVEL_PINCLIENT::RTN_Size (RTN rtn)
- string LEVEL_PINCLIENT::RTN_FindNameByAddress (ADDRINT address)
- RTN LEVEL_PINCLIENT::RTN_FindByAddress (ADDRINT address)
- RTN LEVEL_PINCLIENT::RTN_FindByName (IMG img, const CHAR *name)
- VOID LEVEL_PINCLIENT::RTN_Open (RTN rtn)
- VOID LEVEL_PINCLIENT::RTN_Close (RTN rtn)
- INS LEVEL_PINCLIENT::RTN_InsHead (RTN rtn)
- VOID LEVEL_PINCLIENT::RTN_InsertCall (RTN rtn, IPOINT action, AFUNPTR funptr,...)
- ADDRINT LEVEL_PINCLIENT::RTN_Address (RTN rtn)
- VOID LEVEL_PINCLIENT::RTN_ReplaceWithUninstrumentedRoutine (RTN replacedRtn, AFUNPTR replacementFun)

Descripción Detallada

Una RTN representa las funciones/rutinas/procedimientos producidos por un compilador de un lenguaje de programación procedimental como C. Pin encuentra las rutinas usando la información de la tabla de símbolos. Se debe llamar a PIN_InitSymbols() para que la información de la tabla de símbolos esté disponible. Puede ser accedida en tiempo de instrumentación y en tiempo de análisis.

Formas de iteración:

```
// Paso progresivo por todas las rutinas de una sección.
for( RTN rtn= SEC_RtnHead(sec); RTN_Valid(rtn); rtn = RTN_Next(rtn) )

// Paso regresivo por todas las rutinas de una sección.
for( RTN rtn= SEC_RtnTail(sec); RTN_Valid(rtn); rtn = RTN_Prev(rtn) )
```

Documentación de tipos

- typedef VOID(* LEVEL_PINCLIENT::RTN_INSTRUMENT_CALLBACK)(RTN rtn, VOID *v)
  Función de retorno para instrumentar las rutinas.

Documentación de funciones

- *VOID RTN_AddInstrumentFunction(RTN_INSTRUMENT_CALLBACK fun, VOID *val)*
  Añade una función para instrumentar una rutina.

  Parámetros:

  fun Función añadida.

  val Valor pasado a fun cuando es llamada

- *ADDRINT RTN_Address(RTN rtn)*
  Devuelve:

  La dirección de memoria de la rutina rtn

- *BBL RTN_BblHead(RTN x)*
  Devuelve:

  El primer bloque básico (bbl) de la rutina x.

- *BBL RTN_BblTail(RTN x)*
  Devuelve:

  Último bloque básico (bbl) de la rutina x.

- *VOID RTN_Close(RTN rtn)*
  Cierra rtn, debe ser llamada antes de abrir una nueva rutina.

- *AFUNPTR RTN_ConvertToFunptr(RTN x)*
  Convierte una rutina x en una función.

- *RTN RTN_FindByAddress(ADDRINT address)*
  Parámetros:

  Address Dirección de memoria que se corresponde con la rutina RTN

  Devuelve:

  Busca una rutina a partir de su dirección de memoria. Devuelve el apuntador (handle) de la rutina si la encuentra Si no la encuentra devuelve RTN_Invalid().

  Nota:

  En un programa multihilo puede haber problemas de concurrencia: el apuntador (handle) de la rutina RTN, podría corromperse si otro hilo no cargó el objeto compartido que contenía a la rutina. Para evitar esto usar los cerrojos (PIN_LockClient() antes de llamar a esta función y PIN_UnlockClient() después del último uso del "handle" devuelto. Esto no es necesario si la llamada a esta función es desde una rutina de instrumentación ya que en este caso el "locking‘" es automático).

  Si solo queremos el nombre de la rutina es preferible llamar a RTN_FindNameByAddress, que automáticamente hace el "locking" y devuelve un "string" con el nombre, que no será corrompido si la librería compartida no está cargada.

- *RTN RTN_FindByName(IMG img, const CHAR *name)*
  Parámetros:

  Img Imagen en la cual se buscará la rutina

  Name Nombre de la rutina a buscar en IMG

  Devuelve:

  El "handle" a la rutina encontrada. Si no la encuentra devuelve RTN_Invalid()

- *string RTN_FindNameByAddress(ADDRINT address)*
  Parámetros:

  Address Dirección de memoria que se corresponde con la rutina RTN

  Devuelve:

  El nombre de la rutina ó "" si no la encuentra.

- *VOID RTN_InsertCall(RTN rtn, IPOINT action, AFUNPTR funptr, ...)*
  Inserta una llamada a una función en relación a una rutina.

  Parámetros:

  rtn Rutina a instrumentar

  action Usar IPOINT_BEFORE para llamar a la función antes de la ejecución, o IPOINT_AFTER para llamarla inmediatamente después

  funptr Función de análisis a llamar

  ... Argumentos que recibe funptr

- *INS RTN_InsHead(RTN rtn)*
  Se debe llamar a RTN_Open antes de llamar a esta función por primera vez.

  Devuelve:

  La primera instrucción de una rutina o RTN_Invalid() si no hay instrucciones

- *RTN RTN_Invalid()*
  Devuelve:

  Un valor de RTN que indica que no es una imagen válida (error).

- *const string& RTN_Name(RTN x)*
  Devuelve:

  El nombre de la rutina

- *RTN RTN_Next(RTN x)*
  Devuelve:

  La rutina que sigue a x, o RTN_Invalid() si x es la última en la sección.

- *INT32 RTN_No(RTN x)*
  Devuelve:

  Número de la rutina x.

- *VOID RTN_Open(RTN rtn)*
  Abre rtn, debe ser llamada antes que RTN_InsHead

- *RTN RTN_Prev(RTN x)*
  Devuelve:

  Rutina que precede a x, o RTN_Invalid() si x es la primera en la sección

- *VOID RTN_ReplaceWithUninstrumentedRoutine(RTN replacedRtn, AFUNPTR replacementFun)*
  Sustituye una función en la aplicación (replacedRtn) por otra función ya definida en la Pintool (replacementFun). Esta función de reemplazamiento no es instrumentada.

- *SEC RTN_Sec(RTN x)*
  Devuelve:

  Sección que contiene la rutina x

- *USIZE RTN_Size(RTN rtn)*
  Devuelve:

  El tamaño de la rutina en bytes

- *BOOL RTN_Valid(RTN x)*
  Devuelve:

  True si x no es RTN_Invalid()

B.4. API de instrumentación

Tipos definidos

- typedef VOID(* LEVEL_PINCLIENT::INS_INSTRUMENT_CALLBACK)(INS ins, VOID *v)

Funciones

- VOID LEVEL_PINCLIENT::INS_AddInstrumentFunction (INS_INSTRUMENT_CALLBACK fun, VOID *val)
- VOID LEVEL_PINCLIENT::INS_InsertPredicatedCall (INS ins, IPOINT ipoint, AFUNPTR funptr,...)
- VOID LEVEL_PINCLIENT::INS_InsertCall (INS ins, IPOINT action, AFUNPTR funptr,...)
- VOID LEVEL_PINCLIENT::INS_InsertIfCall (INS ins, IPOINT action, AFUNPTR funptr,...)
- VOID LEVEL_PINCLIENT::INS_InsertThenCall (INS ins, IPOINT action, AFUNPTR funptr,...)

Descripción detallada

Use estas funciones para instrumentar instrucciones.

Documentación de los tipos definidos

- typedef VOID(* LEVEL_PINCLIENT::INS_INSTRUMENT_CALLBACK)(INS ins, VOID *v)
  Función de retorno utilizada para instrumentar instrucciones

Documentación de funciones

- VOID INS_AddInstrumentFunction(INS_INSTRUMENT_CALLBACK fun, VOID *val)

  Añade un función utilizada para instrumentar con granularidad de instrucción

  Parámetros:

  fun Función de instrumentación para instrucciones

  val Pasado como segundo argumento a la función de instrumentación

- VOID INS_InsertCall(INS ins, IPOINT action, AFUNPTR funptr, ...)

  Inserta una llamada a funptr relativa a ins.

  Parámetros:

  ins Instrucción a instrumentar

  action Especifica antes, después, etc. IPOINT_BEFORE es válida para todas las instrucciones. IPOINT_AFTER sólo vale cuando existe un "fall-through" (e.g. Llamadas y saltos incondicionales fallarán). IPOINT_TAKEN_BRANCH no vale para las instrucciones que no sean saltos.

  funptr Inserta una llamada a funptr

  ... Lista de argumentos que se le pasan a funptr. Vea IARG_TYPE, terminado con IARG_END

- VOID INS_InsertIfCall(INS ins, IPOINT action, AFUNPTR funptr, ...)

  Inserta una llamada a funptr relativa a ins y le pasa el resultado al inmediatamente siguiente "then" llamada de análisis.

  Parámetros:

  ins Instrucción a instrumentar

  action Especifica antes, después, etc. IPOINT_BEFORE es válida para todas las instrucciones. IPOINT_AFTER sólo vale cuando existe un "fall-through" (e.g. Llamadas y saltos incondicionales fallarán). IPOINT_TAKEN_BRANCH no vale para las instrucciones que no sean saltos.

  funptr Inserta una llamada a funptr

  ... Lista de argumentos que se le pasan a funptr. Vea IARG_TYPE, terminado con IARG_END

- VOID INS_InsertPredicatedCall(INS ins, IPOINT ipoint, AFUNPTR funptr, ...)

  Vea INS_InsertCall. Cuando una función tiene un predicado y el predicado es false, la función de análisis no es llamada.

- VOID INS_InsertThenCall(INS ins, IPOINT action, AFUNPTR funptr, ...)

  Inserta una llamada a funptr relativa a ins, que será invocado sólo si el inmediatamente anterior "if" de la llamada de análisis devuelve un valor distinto a 0.

  Parámetros:

  ins Instrucción a instrumentar

  action Especifica antes, después, etc. IPOINT_BEFORE es válida para todas las instrucciones. IPOINT_AFTER sólo vale cuando existe un "fall-through" (e.g. Llamadas y saltos incondicionales fallarán). IPOINT_TAKEN_BRANCH no vale para las instrucciones que no sean saltos.

  funptr Inserta una llamada a funptr

  ... Lista de argumentos que se le pasan a funptr. Vea IARG_TYPE, terminado con IARG_END

B.5. API de control genérico

Enumeraciones

- enum LEVEL_CORE::MEMORY_TYPE MEMORY_TYPE_READ, MEMORY_TYPE_WRITE, MEMORY_TYPE_READ2

Funciones

- INT32 LEVEL_CORE::INS_Category (const INS ins)
- USIZE LEVEL_CORE::INS_MemoryWriteSize (INS ins)
- USIZE LEVEL_CORE::INS_MemoryWriteSizeWithoutPrefix (INS ins)
- USIZE LEVEL_CORE::INS_MemoryReadSize (INS ins)
- USIZE LEVEL_CORE::INS_MemoryReadSizeWithoutPrefix (INS ins)
- BOOL LEVEL_CORE::INS_IsMemoryRead (INS ins)
- BOOL LEVEL_CORE::INS_IsMemoryWrite (INS ins)
- BOOL LEVEL_CORE::INS_HasMemoryRead2 (INS ins)
- BOOL LEVEL_CORE::INS_HasFallThrough (INS ins)
- BOOL LEVEL_CORE::INS_IsSyscall (INS ins)
- string LEVEL_CORE::OPCODE_StringShort (UINT32 opcode)
- string LEVEL_CORE::INS_Mnemonic (INS ins)
- BOOL LEVEL_CORE::INS_IsBranchOrCall (INS ins)
- BOOL LEVEL_CORE::INS_IsPcMaterialization (INS ins)
- BOOL LEVEL_CORE::INS_IsCall (INS ins)
- BOOL LEVEL_CORE::INS_IsFarCall(INS ins)
- BOOL LEVEL_CORE::INS_IsProcedureCall (INS ins)
- BOOL LEVEL_CORE::INS_IsRet (INS ins)
- BOOL LEVEL_CORE::INS_IsPrefetch (INS ins)
- BOOL LEVEL_CORE::INS_IsAtomicUpdate (const INS ins)
- BOOL LEVEL_CORE::INS_IsAdjustStackFrame (const INS ins, INT32 &diff)
- BOOL LEVEL_CORE::INS_IsIndirectBranchOrCall (INS ins)
- BOOL LEVEL_CORE::INS_IsDirectBranchOrCall (INS ins)
- BOOL LEVEL_CORE::INS_IsRewritableMemOpBase (INS ins, MEMORY_TYPE mtype, REG &base)
- REG LEVEL_CORE::INS_RegR (INS x, UINT32 k)
- REG LEVEL_CORE::INS_RegW (INS x, UINT32 k)
- OPCODE LEVEL_CORE::INS_Opcode(INS ins)
- string LEVEL_CORE::CATEGORY_StringShort (UINT32 num)
- UINT32 LEVEL_CORE::INS_MaxNumRRegs (INS x)

- UINT32 LEVEL_CORE::INS_MaxNumWRegs (INS x)
- BOOL LEVEL_CORE::INS_RegRContain (const INS ins, const REG reg)
- BOOL LEVEL_CORE::INS_RegWContain(const INS ins, const REG reg)
- BOOL LEVEL_CORE::INS_IsStackRead (const INS ins)
- BOOL LEVEL_CORE::INS_IsStackWrite (const INS ins)
- BOOL LEVEL_CORE::INS_IsIpRelRead (const INS ins)
- BOOL LEVEL_CORE::INS_IsIpRelWrite (const INS ins)
- PREDICATE LEVEL_CORE::INS_GetPredicate (INS ins)
- BOOL LEVEL_CORE::INS_IsPredicated (INS ins)
- BOOL LEVEL_CORE::INS_IsOriginal (INS ins)
- string LEVEL_CORE::INS_Disassemble (INS ins)
- UINT32 LEVEL_CORE::INS_OperandCount (INS ins)
- BOOL LEVEL_CORE::INS_OperandIsMemory (INS ins, UINT32 n)
- BOOL LEVEL_CORE::INS_OperandIsReg (INS ins, UINT32 n)
- REG LEVEL_CORE::INS_OperandReg (INS ins, UINT32 n)
- BOOL LEVEL_CORE::INS_OperandIsImmediate (INS ins, UINT32 n)
- UINT64 LEVEL_CORE::INS_OperandImmediate (INS ins, UINT32 n)
- UINT32 LEVEL_CORE::INS_OperandWidth (INS ins, UINT32 n)
- RTN LEVEL_PINCLIENT::INS_Rtn (INS x)
- INS LEVEL_PINCLIENT::INS_Next(INS x)
- INS LEVEL_PINCLIENT::INS_Prev (INS x)
- INS LEVEL_PINCLIENT::INS_Invalid ()
- BOOL LEVEL_PINCLIENT::INS_Valid(INS x)
- ADDRINT LEVEL_PINCLIENT::INS_Address (INS ins)
- USIZE LEVEL_PINCLIENT::INS_Size (INS ins)
- ADDRINT LEVEL_PINCLIENT::INS_NextAddress (INS ins)

Variables

- GLOBALCONST UINT32 LEVEL_CORE::VARIABLE_MEMORY_REFERENCE_SIZE =õU

Descripción detallada

Utilice estas funciones para examinar una instrucción. Funcionan en todos los juegos de instrucciones.

Documentación de los tipos de enumeración

- enum LEVEL_CORE::MEMORY_TYPE

  Devuelve: true si la instrucción tiene un memory op que pueda ser reescrito. Excluye los accesos a memoria con base en ip o sp, o absolutos. Excluye una escritura si tiene el mismo registro base que una lectura

Documentación de las funciones

- string CATEGORY_StringShort(UINT32 num)

  Devuelve: String de la categoría

- ADDRINT INS_Address (INS ins)

  Devuelve: Dirección de la instrucción

- CATEGORY LEVEL_CORE::INS_Category (const INS ins)

  Devuelve: Categoría de la instrucción. Utilice CATEGORY_StringShort para convertirlo en un string

- string LEVEL_CORE::INS_Disassemble (INS ins)

  Listado del tipo de desensamblado de la instrucción

- PREDICATE LEVEL_CORE::INS_GetPredicate (INS ins)

  Devuelve: Predicado para la instrucción (ia32 es PREDICATE_ALWAYS_TRUE)

- BOOL LEVEL_CORE::INS_HasFallThrough (INS ins)

  Devuelve: true si el tipo de la instrucción tiene un fallthrough path basado en el opcode

- BOOL LEVEL_CORE::INS_HasMemoryRead2 (INS ins)

  Devuelve: true si la instrucción tiene 2 operandos de lectura en memoria

- INS INS_Invalid ( )

  Devuelve: Indica no instrucción

- BOOL INS_IsAdjustStackFrame (const INS ins, INT32 & diff)

  Devuelve: true si ins se ajusta al marco de la pila diff = new sp - old sp

- BOOL LEVEL_CORE::INS_IsAtomicUpdate (const INS ins)

  Devuelve: true si la instrucción puede realizar una actualización atómica de memoria. En ia32, esto es xchg, o una instrucción con el lock prefix set

- BOOL LEVEL_CORE::INS_IsBranchOrCall (INS ins)

  Devuelve: true si ins es una instrucción de salto o llamada, incluyendo tanto tipos directos como indirectos de instrucción

- BOOL LEVEL_CORE::INS_IsCall (INS ins)

  Devuelve: true si ins es una instrucción de llamada

- BOOL LEVEL_CORE::INS_IsDirectBranchOrCall (INS ins)

  Devuelve: true si la dirección objetivo es un desplazamiento del puntero de instrucciones o si es un inmediato

- BOOL INS_IsFarCall (INS ins)

  Devuelve: true si ins es una instrucción de llamada larga (Far Call)

- BOOL LEVEL_CORE::INS_IsIndirectBranchOrCall (INS ins)

  Devuelve: true si el objetivo del salto viene de registro o de memoria

- BOOL LEVEL_CORE::INS_IsIpRelRead (const INS ins)

  Es una lectura relativa a IP

- BOOL LEVEL_CORE::INS_IsIpRelWrite (const INS ins)

  Es un escritura relativa a IP

- BOOL LEVEL_CORE::INS_IsMemoryRead (INS ins)

  Devuelve: true si la instrucción lee de memoria

- BOOL LEVEL_CORE::INS_IsMemoryWrite (INS ins)

  Devuelve: true si la instrucción escribe en memoria

- BOOL INS_IsOriginal (INS ins)

  Devuelve: true si es una instrucción original

- BOOL INS_IsPcMaterialization (INS ins)

  Devuelve: true si es una llamada a la siguiente instrucción, que es un truco para materializar el puntero de instrucciones

- BOOL LEVEL_CORE::INS_IsPredicated (INS ins)

- BOOL LEVEL_CORE::INS_IsPrefetch (INS ins)

  Devuelve: true si esta instrucción es una prebúsqueda

- BOOL LEVEL_CORE::INS_IsProcedureCall (INS ins)

  Devuelve: true si ins es una llamada a procedimiento. Esto filtra las instrucciones de llamada que se utilizan para otros propósitos

- BOOL LEVEL_CORE::INS_IsRet (INS ins)

  Devuelve: true si ins es una instrucción de retorno

- BOOL LEVEL_CORE::INS_IsRewritableMemOpBase (INS ins, MEMORY_TYPE mtype, REG & base)

  Devuelve: true si la instrucción tiene un memory op que puede ser sobreescrito. Excluye los accesos a memoria con base en ip o sp, o absolutos. Excluye también si una escritura tiene el mismo registro base que una lectura

- BOOL LEVEL_CORE::INS_IsStackRead (const INS ins)

  Devuelve: true si ins es una lectura de la pila

- BOOL LEVEL_CORE::INS_IsStackWrite (const INS ins)

  Devuelve: true si ins es una escritura en la pila

- BOOL LEVEL_CORE::INS_IsSyscall (INS ins)

  Devuelve: true si la instrucción es una llamada al sistema

- UINT32 LEVEL_CORE::INS_MaxNumRRegs (INS x)

  Devuelve: Máximo número de operandos de lectura

- UINT32 LEVEL_CORE::INS_MaxNumWRegs (INS x)

  Devuelve: Máximo número de operandos de escritura

- UINT32 LEVEL_CORE::INS_MemoryReadSize (INS ins)

  Devuelve: el tamaño de la lectura de memoria en bytes o VARIABLE_MEMORY_REFERENCE_SIZE si el tamaño se ha determinado en tiempo de ejecución (e.g. movs en ia32)

- USIZE INS_MemoryReadSizeWithoutPrefix (INS ins)

  Devuelve: Tamaño de la lectura de memoria en bytes, ignorando todo prefijo de repetición

- UINT32 LEVEL_CORE::INS_MemoryWriteSize (INS ins)

  Devuelve: el tamaño de la escritura de memoria en bytes o VARIABLE_MEMORY_REFERENCE_SIZE si el tamaño se determina en tiempo de ejecución (e.g. movs en ia32)

- USIZE INS_MemoryWriteSizeWithoutPrefix (INS ins)

  Devuelve: Tamaño de la escritura en memoria en bytes, ignorando todo prefijo de repetición

- string LEVEL_CORE::INS_Mnemonic (INS ins)

  Devuelve: String del código mnemotécnico

- INS INS_Next (INS x)

  Devuelve: Instrucción que sigue a x, o INS_Invalid() si x es la última en la rutina o traza

- ADDRINT INS_NextAddress (INS ins)

  Obtiene la dirección de la siguiente instrucción. Para Itanium, no tiene sentido añadir el tamaño a la dirección para conseguir la siguiente instrucción, así que use esta primitiva para todas las arquitecturas.

  Devuelve: Dirección de la instrucción siguiente

- OPCODE LEVEL_CORE::INS_Opcode (INS ins)

  Devuelve: Opcode de la instrucción. Use INS_Mnemonic si quiere un string

- UINT32 INS_OperandCount (INS ins)

  Devuelve: número de operandos

- UINT64 INS_OperandImmediate (INS ins, UINT32 n)

  Devuelve: valor inmediatos para el operando

- BOOL INS_OperandIsImmediate (INS ins, UINT32 n)

  Devuelve: true si este operando es un inmediato

- BOOL INS_OperandIsMemory (INS ins, UINT32 n)

  Devuelve: true si este operando es una referencia a memoria

- BOOL INS_OperandIsReg (INS ins, UINT32 n)

  Devuelve: true si este operando es un inmediato

- REG INS_OperandReg (INS ins, UINT32 n)

  Devuelve: registro para este inmediato

- UINT32 INS_OperandWidth (INS ins, UINT32 n)

  Devuelve: anchura del operando en bits

- INS INS_Prev (INS x)

  Devuelve: Instrucción que precede a x, oINS_Invalid() si x es la primera de la rutina o traza

- REG LEVEL_CORE::INS_RegR (INS ins, UINT32 n)

  Devuelve: k-ésimo registro de lectura de la instrucción x, incluyendo lecturas implícitas (e.g. el puntero de la pila es leído por push en ia32)

- BOOL INS_RegRContain (const INS ins, const REG reg)

  Devuelve: true si ins utiliza reg como un operando de lectura

- REG LEVEL_CORE::INS_RegW (INS ins, UINT32 n)

  Devuelve: k-ésimo registro de escritura de la instrucción x, incluyendo escrituras implícitas (e.g. el puntero de la pila es escrito con push en ia32)

- BOOL INS_RegWContain (const INS ins, const REG reg)

  Devuelve: true si ins utiliza reg como un operando de escritura

- RTN INS_Rtn (INS x)

  Devuelve: Rutina que contiene esta instrucción

- USIZE INS_Size (INS ins)

  Devuelve: Tamaño de la instrucción en bytes

- BOOL INS_Valid (INS x)

  Devuelve: True si x no es INS_Invalid()

- string LEVEL_CORE::OPCODE_StringShort (UINT32 opcode)

  Devuelve: String con el opcode de la instrucción

Documentación de la variable

- GLOBALCONST UINT32 LEVEL_CORE::VARIABLE_MEMORY_REFERENCE_SIZE=0̃U

  Valor devuelto por INS_MemoryReadSize oINS_MemoryWriteSize cuando la instrucción lee/escribe un área de memoria de tamaño variable. (e.g. stos en ia32). Use IARG_MEMORYREAD_SIZE/IARG_MEMORYWRITE_SIZE para determinar el valor

B.6. TRAZA: Entrada Simple, Secuencia de instrucciones con múltiples salidas

Tipos Definidos

- typedef const TRACE_CLASS * LEVEL_PINCLIENT::TRACE
- typedef VOID(*LEVEL_PINCLIENT::TRACE_INSTRUMENT_CALLBACK)(TRACE trace, VOID *v)

Funciones

- BBL LEVEL_PINCLIENT::TRACE_AddInlineReturnEdg (TRACE trace)
- BBL LEVEL_PINCLIENT::TRACE_AddInlineCallEdg (TRACE trace)
- BBL LEVEL_PINCLIENT::TRACE_AddBranchEdg (TRACE trace)
- BBL LEVEL_PINCLIENT::TRACE_AddFallthroughEdg (TRACE trace)
- VOID LEVEL_PINCLIENT::TRACE_StraightenControlFlow (TRACE trace)
- ADDRINT LEVEL_PINCLIENT::TRACE_GenerateCode (TRACE trace)
- VOID LEVEL_PINCLIENT::TRACE_AddInstrumentFunction (TRACE_INSTRUMENT_CALLBACK fun, VOID *val)
- VOID LEVEL_PINCLIENT::TRACE_InsertCall (TRACE trace, IPOINT action, AFUNPTR funptr,...)
- VOID LEVEL_PINCLIENT::TRACE_InsertIfCall (TRACE trace, IPOINT action,AFUNPTR funptr,...)
- VOID LEVEL_PINCLIENT::TRACE_InsertThenCall (TRACE trace, IPOINT action, AFUNPTR funptr,...)
- BBL LEVEL_PINCLIENT::TRACE_BblHead (TRACE trace)
- BBL LEVEL_PINCLIENT::TRACE_BblTail (TRACE trace)
- BOOL LEVEL_PINCLIENT::TRACE_Original (TRACE trace)
- ADDRINT LEVEL_PINCLIENT::TRACE_Address (TRACE trace)
- USIZE LEVEL_PINCLIENT::TRACE_Size (TRACE trace)
- RTN LEVEL_PINCLIENT::TRACE_Rtn (TRACE trace)
- BOOL LEVEL_PINCLIENT::TRACE_HasFallThrough (TRACE trace)
- UINT32 LEVEL_PINCLIENT::TRACE_NumBbl (TRACE trace)
- UINT32 LEVEL_PINCLIENT::TRACE_NumIns (TRACE trace)

Descripción Detallada

La instrumentación de trazas permite a Pin inspeccionar e instrumentar una traza ejecutable en un solo paso. Las trazas suelen comenzar en el objetivo de un salto (el que se haya elegido) y terminan con un salto incondicional, incluyendo llamadas y retornos. Pin garantiza que una traza solo tiene una entrada al principio pero puede contener múltiples salidas. Si hay un salto en medio de la traza, Pin construye una nueva traza que comienza con el objetivo de dicho salto. Pin divide la traza en bloques básicos (BBLs). Un BBL es una secuencia de instrucciones con una sola entrada y una sola salida. Los saltos en medio de un bbl comienzan una nueva traza y por tanto un nuevo bbl. También hay llamadas de análisis para un bbl e incluso para cada instrucción. Reduciendo el numero de llamadas de análisis se hace más eficiente la instrumentación.

Secuencia de instrucciones que se ejecutan en orden y que puede tener múltiples salidas. Si Pin detecta un salto a una instrucción en el medio de una traza, creará una nueva traza que comenzará en el destino del salto. Ver Instrumentation Granularity.

Formas de iteración: // Paso progresivo por todos los bloques básicos de una traza. for(BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))

Documentación de Tipos

- typedef const TRACE_CLASS* LEVEL_PINCLIENT::TRACE

  Estructura que almacena una traza

- typedef VOID(* LEVEL_PINCLIENT::TRACE_INSTRUMENT_CALLBACK)(TRACE trace, VOID *v)

  Función de retorno usada para instrumentar trazas. Documentación de

Funciones

- BBL TRACE_AddBranchEdg(TRACE trace)

  Si la última instrucción de la traza es un salto directo entonces se añade el bbl de la ruta objetivo como la ruta del fallo(fallthrough) en la traza.. TRACE_StraightenControlFlow debe ser invocada antes de compilar la traza .

  Nota : el fallthrough indica el final de la traza.

  Parámetros:

  trace Traza a añadir al nuevo bloque básico

  Devuelve:

  bbl Un apuntador (handle) al bloque básico recientemente añadido.

- BBL TRACE_AddFallthroughEdg(TRACE trace)

  Añade el bloque básico de la ruta del fallo de la última instrucción actual en la traza, a la última instrucción de la traza.

Parámetros:

trace Traza a la que añadir el nuevo bloque básico

Devuelve:

bbl Un apuntador (handle) al bloque recientemente añadido.

- BBL TRACE_AddInlineCallEdg(TRACE trace)

  Si la última instrucción de la traza es una llamada directa, entonces inserta el destino de la llamada en la traza.

  Parámetros:

  trace Traza a la que añadir el nuevo bloque básico

  Devuelve:

  bbl Un apuntador (handle) al bloque recientemente añadido

- BBL TRACE_AddInlineReturnEdg(TRACE trace)

  Inserta la ruta devuelta por una llamada que previamente había sido insertada en esta traza usando TRACE_AddInlineCallEdg. Esto requiere que la instrucción que está al final sea del tipo return.

  Parámetros:

  trace Traza a la que añadir el nuevo bbl

  Devuelve:

  bbl Un apuntador (handle) al bbl recientemente añadido

- VOID TRACE_AddInstrumentFunction(TRACE_INSTRUMENT_CALLBACK fun, VOID *val)

  Añade una función para instrumentar la granuralidad de la traza.

  Parámetros:

  fun Función de instrumentación para trazas

  val Argumento de fun

- ADDRINT TRACE_Address (TRACE trace)

  Devuelve:

  Dirección de aplicación de una traza.

- BBL TRACE_BblHead(TRACE trace)

  Devuelve:

  Primer bbl de una traza.

- BBL TRACE_BblTail(TRACE trace)

  Devuelve:

  Último bbl de una traza.

- ADDRINT TRACE_GenerateCode(TRACE trace)

  Compila y sitúa una traza en la cache de código..

  Devuelve:

  Dirección en la que la traza ha sido situada.

- BOOL TRACE_HasFallThrough(TRACE trace)

  Devuelve:

  True si la traza tiene un fallo (fall-through) sino devuelve false.

- VOID TRACE_InsertCall(TRACE trace, IPOINT action, AFUNPTR funptr, ...)

  Inserta una llamada en relación a una traza.

  Parámetros:

  trace Traza a instrumentar

  action Especifica cuando se va a llamar a la función : después, antes, etc. IPOINT_BEFORE antes, válido para todas las instrucciones. IPOINT_AFTER después, solo cuando existe un fall-through (e.g. Llamadas y saltos incondicionales con fallo). IPOINT_ANYWHERE pondrá la instrumentación en lugar de la traza en el que se obtenga mejor rendimiento. IPOINT_TAKEN_BRANCH no es válido si no hay saltos.

  funptr Función de análisis a llamar

  ... Argumentos que se le pasan a funptr

- VOID TRACE_InsertIfCall(TRACE trace, IPOINT action, AFUNPTR funptr, ...)

  Inserta una llamada en relación a una traza. y pasa el resultado a la inmediatamente siguiente rutina"then"de análisis.

  Parámetros:

  trace Traza a instrumentar

  action Especifica cuando se va a llamar a la función : después, antes, etc. IPOINT_BEFORE antes, válido para todas las instrucciones. IPOINT_AFTER después, solo cuando existe un fall-through (e.g. Llamadas y saltos incondicionales con fallo). IPOINT_ANYWHERE pondrá la instrumentación en lugar de la traza en el que se obtenga mejor rendimiento. IPOINT_TAKEN_BRANCH no es válido si no hay saltos.

  funptr Función de análisis a llamr

  ... Argumentos que se le pasan a funptr

- VOID TRACE_InsertThenCall(TRACE trace, IPOINT action, AFUNPTR funptr, ...)

  Inserta una llamada relativa a una traza que será invocada solamente si la rutina previa "if"de análisis devuelve un valor distinto de cero.

  Parámetros:

  trace Traza a instrumentar

  action Especifica cuando se va a llamar a la función : después, antes, etc. IPOINT_BEFORE antes, válido para todas las instrucciones. IPOINT_AFTER después, solo cuando existe un fall-through (e.g. Llamadas y saltos incondicionales con fallo). IPOINT_TAKEN_BRANCH no es válido si no hay saltos

  funptr Función de análisis a llamar

  ... Argumentos que se le pasan a funptr

- UINT32 TRACE_NumBbl (TRACE trace)

  Devuelve:

  El número de bloques básicos en una traza.

- UINT32 TRACE_NumIns(TRACE trace)

  Devuelve:

  Número de instrucciones en una traza.

- BOOL TRACE_Original(TRACE trace)

  Devuelve:

  Si la ha sido instrumentada u optimizada desde que fue creada o no.

- RTN TRACE_Rtn(TRACE trace)

  Devuelve:

  Rutina que contiene la primera instrucción de la traza.

- USIZE TRACE_Size(TRACE trace)

  Devuelve:

  El tamaño del código de la aplicación de una traza.

- VOID TRACE_StraightenControlFlow(TRACE trace)

  Construye el flujo de control para la traza que acabamos de generar; permite que se intercalen las instrucciones insertadas en la traza Debe ser llamada antes de mirar un traza a la que se le haya aplicado la función TRACE_AddInlineCallEdg o TRACE_AddInlineReturnEdg para añadir bbls a la traza. Una vez invocada esta función, el flujo de control de traza está arreglado y se puede ver como estaría en la memoria después de la compilación..

B.7. SYM: Objeto Symbol

Funciones

- SYM LEVEL_PINCLIENT::SYM_Next (SYM x)
- SYM LEVEL_PINCLIENT::SYM_Prev (SYM x)
- const string & LEVEL_PINCLIENT::SYM_Name (SYM x)
- SYM LEVEL_PINCLIENT::SYM_Invalid ()
- BOOL LEVEL_PINCLIENT::SYM_Valid (SYM x)
- BOOL LEVEL_PINCLIENT::SYM_Dynamic (SYM x)
- ADDRINT LEVEL_PINCLIENT::SYM_Value (SYM x)
- UINT32 LEVEL_PINCLIENT::SYM_Index (SYM x)

Descripción detallada

Los SYMs derivan de los registros de símbolos de ELF. Se debe invocar a PIN_InitSymbols para habilitar algunos símbolos. Se puede acceder a ellos durante la instrumentación y durante el análisis.

Uso en la iteración:

// Recorre todos los símbolos de una imagen

for(SYM sym= IMG_RegsymHead(img); SYM_Valid(sym); sym = SYM_Next(sym))

Documentación de las funciones

- BOOL SYM_Dynamic(SYM x)

  Devuelve:

  True si x es un símbolo dinámico

- UINT32 SYM_Index(SYM x)

  Devuelve:

  Índice de la sección de un símbolo

- SYM SYM_Invalid()

  Devuelve:

  Se usa para no indicar símbolo

- const string& SYM_Name(SYM x)

  Devuelve:

  Nombre del símbolo

- SYM SYM_Next(SYM x)

  Devuelve:

  Rutina que sigue a x, o SYM_Invalid() si x es el último en la sección

- SYM SYM_Prev(SYM x)

  Devuelve:

  Rutina que precede a x, o SYM_Invalid() si x es el primero de la sección

- BOOL SYM_Valid(SYM x)

  Devuelve:

  True si x no es SYM_Invalid()

- ADDRINT SYM_Value(SYM x)

  Devuelve:

  Valor del símbolo, normalmente una dirección relativa al principio de la imagen

B.8. REG: Objeto Register (genérico)

Descripción detallada

Se puede acceder a ellos durante la instrumentación y durante el análisis.

Funciones

- BOOL LEVEL_BASE::REG_is_gr (REG reg)
- BOOL LEVEL_BASE::REG_is_fr (REG reg)
- BOOL LEVEL_BASE::REG_is_gr64 (REG reg)
- BOOL LEVEL_BASE::REG_is_gr32 (REG reg)
- BOOL LEVEL_BASE::REG_is_gr16 (REG reg)
- string LEVEL_BASE::REG_StringShort (REG reg)

Documentación de las funciones

- BOOL LEVEL_BASE::REG_is_fr(REG reg)

  Devuelve:

  TRUE si es un registro de punto flotante

- BOOL LEVEL_BASE::REG_is_gr(REG reg)

  Devuelve:

  TRUE si es un registro de propósito general

- BOOL REG_is_gr16(REG reg)

  Devuelve:

  TRUE si es un registro de 16 bits de propósito general

- BOOL REG_is_gr32(REG reg)

  Devuelve:

  TRUE si es un registro de 32 bits de propósito general

- BOOL REG_is_gr64(REG reg)

  Devuelve:

  TRUE si es un registro de 64 bits de propósito general

- string LEVEL_BASE::REG_StringShort(REG reg)

  Convierte un REG en un string

B.9. Argumentos para las rutinas de instrumentación

Tipos enumerados

- enum LEVEL_PINCLIENT::IPOINT { IPOINT_INVALID, LEVEL_PINCLIENT::IPOINT_BEFORE, LEVEL_PINCLIENT::IPOINT_AFTER, LEVEL_PINCLIENT::IPOINT_ANYWHERE, LEVEL_PINCLIENT::IPOINT_TAKEN_BRANCH }

- enum LEVEL_PINCLIENT::IARG_TYPE { IARG_INVALID, LEVEL_PINCLIENT::IARG_END, LEVEL_PINCLIENT::IARG_ADDRINT, LEVEL_PINCLIENT::IARG_PTR, LEVEL_PINCLIENT::IARG_BOOL, LEVEL_PINCLIENT::IARG_UINT32, LEVEL_PINCLIENT::IARG_INST_PTR, LEVEL_PINCLIENT::IARG_REG_VALUE, LEVEL_PINCLIENT::IARG_MEMORYREAD_EA, LEVEL_PINCLIENT::IARG_MEMORYREAD2_EA, LEVEL_PINCLIENT::IARG_MEMORYWRITE_EA, LEVEL_PINCLIENT::IARG_MEMORYREAD_SIZE, LEVEL_PINCLIENT::IARG_MEMORYWRITE_SIZE, LEVEL_PINCLIENT::IARG_BRANCH_TAKEN, LEVEL_PINCLIENT::IARG_BRANCH_TARGET_ADDR, LEVEL_PINCLIENT::IARG_FALLTHROUGH_ADDR, IARG_EXECUTING, IARG_PREDICATE, IARG_REP_SIZE, IARG_STACK, IARG_STACK_ADDR, IARG_SYSCALL_NUMBER, LEVEL_PINCLIENT::IARG_SYSCALL_ARG0, IARG_SYSCALL_ARGBASE = IARG_SYSCALL_ARG0, LEVEL_PINCLIENT::IARG_SYSCALL_ARG1, LEVEL_PINCLIENT::IARG_SYSCALL_ARG2, LEVEL_PINCLIENT::IARG_SYSCALL_ARG3, LEVEL_PINCLIENT::IARG_SYSCALL_ARG4, LEVEL_PINCLIENT::IARG_SYSCALL_ARG5, IARG_SYSCALL_ARGLAST = IARG_SYSCALL_ARG5, LEVEL_PINCLIENT::IARG_SYSCALL_RESULT, LEVEL_PINCLIENT::IARG_G_ARG0_CALLEE, IARG_G_ARGBASE_CALLEE = IARG_G_ARG0_CALLEE, LEVEL_PINCLIENT::IARG_G_ARG1_CALLEE, LEVEL_PINCLIENT::IARG_G_ARG2_CALLEE, LEVEL_PINCLIENT::IARG_G_ARG3_CALLEE, LEVEL_PINCLIENT::IARG_G_ARG4_CALLEE, LEVEL_PINCLIENT::IARG_G_ARG5_CALLEE, IARG_G_ARGLAST_CALLEE = IARG_G_ARG5_CALLEE, LEVEL_PINCLIENT::IARG_G_ARG0_CALLER, IARG_G_ARGBASE_CALLER = IARG_G_ARG0_CALLER, LEVEL_PINCLIENT::IARG_G_ARG1_CALLER, LEVEL_PINCLIENT::IARG_G_ARG2_CALLER, LEVEL_PINCLIENT::IARG_G_ARG3_CALLER, LEVEL_PINCLIENT::IARG_G_ARG4_CALLER, LEVEL_PINCLIENT::IARG_G_ARG5_CALLER, IARG_G_ARGLAST_CALLER = IARG_G_ARG5_CALLER, LEVEL_PINCLIENT::IARG_RETURN_IP, LEVEL_PINCLIENT::IARG_G_RESULT0, IARG_G_RETBASE = IARG_G_RESULT0, LEVEL_PINCLIENT::IARG_G_RESULT1, IARG_G_RESULTLAST = IARG_G_RESULT1, IARG_THREAD_ID, LEVEL_PINCLIENT::IARG_CONTEXT, LEVEL_PINCLIENT::IARG_CHECKPOINT, LEVEL_PINCLIENT::IARG_CONTEXT_TMP, LEVEL_PINCLIENT::IARG_RETURN_REGS }

Funciones

- VOID LEVEL_PINCLIENT::PIN_MakeContext (VOID *handle, CONTEXT *ctxt)
- GLOBALCFUN VOID *LEVEL_PINCLIENT::DoJitThreadStartRoutine (THREAD_STARTROUTINE startroutine, VOID *arg, VOID *threadTag, ADDRINT returnIp, ADDRINT newContextSp)
- VOID *LEVEL_PINCLIENT::PIN_RegisterNewThread ()

Documentación de tipos enumerados

enum LEVEL_PINCLIENT::IARG_TYPE

Determina los argumentos que se pasan a la llamada de análisis
Valores de enumeración:

IARG_END	Debe ser el último en la lista IARGl para InsertCall.
IARG_ADDRINT	Constante de tipo ADDRINT (requerido arg adicional del tipo entero largo).
IARG_PTR	Constante de tipo VOID * (Requerido puntero arg adicional).

IARG_BOOL	Constante de tipo (Requerido arg BOOL adicional).
IARG_UINT32	Constante de tipo UINT32 (additional integer arg required).
IARG_INST_PTR	Puntero de instrucción
IARG_REG_VALUE	Valor de un registro(Requerido arg register adicional).
IARG_MEMORYREAD_EA	Dirección efectiva de una lectura de memoria.
IARG_MEMORYREAD2_EA	Dirección efectiva de una segunda lectura de memoria. (e.g. Segundo operando en cmps en ia32).
IARG_MEMORYWRITE_EA	Dirección efectiva de una escritura en memoria.
IARG_MEMORYREAD_SIZE	Tamaño en bytes de una lectura de memoria.
IARG_MEMORYWRITE_SIZE	Tamaño en bytes de una escritura en memoria.
IARG_BRANCH_TAKEN	Distinto de cero si se realiza un salto
IARG_BRANCH_TARGET_ADDR	Dirección objetivo de esta instrucción de salto
IARG_FALLTHROUGH_ADDR	Dirección de Fall through de esta instrucción
IARG_SYSCALL_ARG0	Primer argumento de una llamada al sistema (solo IPOINT_BEFORE)
IARG_SYSCALL_ARG1	Segundo argumento de una llamada al sistema (solo IPOINT_BEFORE)
IARG_SYSCALL_ARG2	Tercer argumento de una llamada al sistema (solo IPOINT_BEFORE)
IARG_SYSCALL_ARG3	Cuarto argumento de una llamada al sistema (solo IPOINT_BEFORE)
IARG_SYSCALL_ARG4	Quinto argumento de una llamada al sistema (solo IPOINT_BEFORE)
IARG_SYSCALL_ARG5	Sexto argumento de una llamada al sistema (solo IPOINT_BEFORE)
IARG_SYSCALL_RESULT	Valor devuelto por una llamada al sistema (solo IPOINT_AFTER)
IARG_G_ARG0_CALLEE	Primer argumento de un procedimiento no de punto flotante, visto por el receptor.
IARG_G_ARG1_CALLEE	Segundo argumento de un procedimiento no de punto flotante, visto por el receptor.
IARG_G_ARG2_CALLEE	Tercer argumento de un procedimiento no de punto flotante, visto por el receptor.
IARG_G_ARG3_CALLEE	Cuarto argumento de un procedimiento no de punto flotante, visto por el receptor.
IARG_G_ARG4_CALLEE	Quinto argumento de un procedimiento no de punto flotante, visto por el receptor.
IARG_G_ARG5_CALLEE	Sexto argumento de un procedimiento no de punto flotante, visto por el receptor.
IARG_G_ARG0_CALLER	Primer argumento de un procedimiento no de punto flotante, visto por el emisor.
IARG_G_ARG1_CALLER	Segundo argumento de un procedimiento no de punto flotante, visto por el emisor.
IARG_G_ARG2_CALLER	Tercer argumento de un procedimiento no de punto flotante, visto por el emisor.
IARG_G_ARG3_CALLER	Cuarto argumento de un procedimiento no de punto flotante, visto por el emisor.
IARG_G_ARG4_CALLER	Quinto argumento de un procedimiento no de punto flotante, visto por el emisor.
IARG_G_ARG5_CALLER	Sexto argumento de un procedimiento no de punto flotante, visto por el emisor.
IARG_RETURN_IP	Dirección devuelta por una llamada a una función, válida sólo en el punto de entrada de la función.
IARG_G_RESULT0	Primer valor devuelto por el procedimiento no de punto flotante.
IARG_G_RESULT1	Segundo valor devuelto por el procedimiento no de punto flotante.
IARG_CONTEXT	Apuntador para acceder a un contexto completo.
IARG_CHECKPOINT	Apuntador para acceder a un çheckpoint"(estado del procesador).
IARG_CONTEXT_TMP	Apuntador para acceder a un contexto (estado de la arquitectura).
IARG_RETURN_REGS	(Requerido arg register adicional)

enum LEVEL_PINCLIENT::IPOINT

Determinan donde es situada la llamada de análisis en función del objeto instrumentado.
Valores de enumeración:

IPOINT_BEFORE	Inserta una llamada antes de una rutina de instrumentación.
IPOINT_AFTER	Inserta una llamada en la ruta del fallthrough de una instrucción o devuelve la ruta de una rutina.
IPOINT_ANYWHERE	Inserta una llamada en cualquier lugar dentro de una traza o un bbl.
IPOINT_TAKEN_BRANCH	Inserta una llamada en el limite del salto.

Documentación de Funciones

- GLOBALCFUN VOID* DoJitThreadStartRoutine(THREAD_STARTROUTINE startroutine, VOID *arg, VOID *threadTag, ADDRINT returnIp, ADDRINT newContextSp)
  Cuando se crea un nuevo hilo , pide permiso a Pin para comenzar la rutina jit (just in time) y luego la transfiere el control. arg es el argumento para startroutine. Después de que acabe start_routine, el control retornará al emisor de PIN_JitThreadStartRoutine().

- VOID PIN_MakeContext(VOID * handle, CONTEXT * ctxt)
  Dado un apuntador, lo sitúa en un contexto. El apuntador (handle) viene de IARG_CONTEXT. Ver IARG_TYPE. Debería ser llamada justo en la entrada de la rutina.

- VOID* PIN_RegisterNewThread()
  Registra un hilo nuevo creado con Pin . Devuelve una etiqueta para el hilo.

B.10. API de Optimización

Funciones

- BOOL LEVEL_PINCLIENT::INS_IsPinXfer (INS ins)
- BOOL LEVEL_PINCLIENT::INS_IsNativeXfer (INS ins)
- VOID LEVEL_PINCLIENT::INS_SetNativeXfer (INS ins)
- VOID LEVEL_PINCLIENT::INS_SetPinXfer (INS ins)
- BOOL LEVEL_PINCLIENT::INS_IsNativeCall (INS ins)
- BOOL LEVEL_PINCLIENT::INS_IsXlateCall (INS ins)
- VOID LEVEL_PINCLIENT::INS_SetXlateCall (INS ins)
- VOID LEVEL_PINCLIENT::INS_SetNativeCall (INS ins)
- VOID LEVEL_PINCLIENT::INS_RedirectControlFlowToAddress (INS ins, ADDRINT target_addr)

Descripción detallada

Esta API permite al usuario modificar el programa para su optimización.

Documentación de Funciones

- BOOL INS_IsNativeCall(INS ins)

  Devuelve:

  TRUE si una llamada está marcada como para ser ejecutada de la forma original (nativa).

- BOOL INS_IsNativeXfer(INS ins)

  Devuelve:

  TRUE si una instrucción de flujo de control ha sido implementada para retornar el control al código original en la ruta tomada. Este es el mecanismo por defecto.

- BOOL INS_IsPinXfer(INS ins)

  Devuelve:

  TRUE si una instrucción de flujo de control ha sido implementada para transferir el control al generador de trazas sobre la ruta tomada.

- BOOL INS_IsXlateCall(INS ins)

  Devuelve:

  TRUE si se marca una instrucción de llamada para traducirla como un .ªpilar y salto al objetivo" para asegurar la transparencia de la aplicación con respecto a donde se originó la llamada.

- VOID INS_RedirectControlFlowToAddress(INS ins, ADDRINT target_addr)

  Asigna un salto determinado o una instrucción de llamada a la dirección dada.

  Parameters:

  ins El salto o la instrucción de llamada a parchear con la nueva dirección

  dst_addr La dirección a codificar como el nuevo destino del salto/llamada.

- VOID INS_SetNativeCall(INS ins)

  Una instrucción de llamada no se traduce como un .ªpilar y salto". Esto implica que la aplicación podría darse cuenta de que la traza es la que está realizando la llamada en la cache de código y de esta forma cualquier código que dependa de la dirección devuelta en el lugar objetivo no sería válido por más tiempo, lo que llevaría a la corrupción del programa. Hacer esto implica que cualquier instrucción que siga a la llamada en la traza, es ejecutada.

- VOID INS_SetNativeXfer(INS ins)

  Coloca una instrucción de flujo de control para retornar el control al código original en la ruta tomada. Este es el mecanismo por defecto.

- VOID INS_SetPinXfer(INS ins)

  Coloca una instrucción de flujo de control para retornar el control al generador de trazas sobre la ruta tomada.

- VOID INS_SetXlateCall(INS ins)

  Una instrucción de llamada se traduce por un return una dirección + apilar y saltar" para asegurarse que la aplicación no se da cuenta de que la llamada ocurre en la cache de código si miró a la dirección devuelta en la pila mientras estábamos colocando la dirección, devuelta por el código original, en la pila.. Esto implica que cuando un return se ejecuta en el destino de la llamada, el control vuelve a la aplicación original. Cualquier instrucción de fallo (fallthrough) en la traza no será ejecutada. Ver también INS_SetNativeCall.

B.11. API para los puntos de control (checkpoints)

Funciones

- VOID LEVEL_PINCLIENT::PIN_SaveCheckpoint (CHECKPOINT *chkptFrom, CHECKPOINT *chkptTo)
- VOID LEVEL_PINCLIENT::PIN_Resume (CHECKPOINT *chkpt)
- VOID LEVEL_PINCLIENT::PIN_SetContextReg (CONTEXT_TMP *ctxt, const REG reg, const ADDRINT val)
- ADDRINT LEVEL_PINCLIENT::PIN_GetContextReg (CONTEXT_TMP *ctxt, const REG reg)
- VOID LEVEL_PINCLIENT::PIN_SaveContext (CONTEXT_TMP *ctxtFrom, CONTEXT_TMP *ctxtTo)
- VOID LEVEL_PINCLIENT::PIN_ExecuteAt (CONTEXT_TMP *ctxt)

Descripción detallada

API para guardar el estado del procesador para la reejecución del programa: Un CHECKPOINT es el estado real del procesador, mientras que CONTEXT_TMP es el estado de la arquitectura. Use PIN_Resume(CHECKPOINT*) cuando vuelva dinámicamente a un punto de ejecución anterior en ejecución, y use PIN_ExecuteAt(CONTEXT_TMP*) cuando empiece en un punto arbitrario (o cuando el usuario quiera introducir valores de registros específicos). El usuario es responsable de vigilar el estado que no sea del procesador (memory, etc.) en la PIN tool. Nota: CHECKPOINT & CONTEXT_TMP no guardan el estado del punto flotante.

Documentación de las funciones

- VOID PIN_ExecuteAt(CONTEXT_TMP *ctxt)

  Comienza la ejecución en un punto arbitrario dado un estado de la arquitectura.

- ADDRINT PIN_GetContextReg(CONTEXT_TMP *ctxt, const REG reg)

  Devuelve el valor del registro almacenado en el contexto

- VOID PIN_Resume(CHECKPOINT *chkpt)

  Reanuda la ejecución en el punto de control guardado (estado del procesador). (Nota: no se puede volcar la cache de código entre PIN_SaveCheckpoint y PIN_Resume - NYI)

- VOID PIN_SaveCheckpoint(CHECKPOINT *chkptFrom, CHECKPOINT *chkptTo)

  Copia el punto de control 'chkptFrom' a 'chkptTo'

- VOID PIN_SaveContext(CONTEXT_TMP *ctxtFrom, CONTEXT_TMP *ctxtTo)

  Copia el contexto 'ctxtFrom' a 'ctxtTo'

- VOID PIN_SetContextReg(CONTEXT_TMP *ctxt, const REG reg, const ADDRINT val)

  Fija el registro del contexto al valor indicado

B.12. LOCK: Primitivas de locking

Tipos definidos

- typedef INT32 LEVEL_BASE::PIN_LOCK

Enumeraciones

- enum { KEY_USER_MIN = 140, KEY_USER_MAX = 190, KEY_OS_TID = 198, KEY_THREADID = 199, KEY_LAST = 200 }

Funciones

- VOID LEVEL_BASE::InitLock (PIN_LOCK *lock)
- VOID LEVEL_BASE::GetLock (PIN_LOCK *lock, INT32 val)
- INT32 LEVEL_BASE::ReleaseLock (PIN_LOCK *lock)

Descripción detallada

Primitivas para locking

Documentación de tipo definidos

- typedef INT32 LEVEL_BASE::PIN_LOCK

  Spinlock

Documentación de las funciones

- VOID GetLock(PIN_LOCK *lock, INT32 val)

  Obtener el cerrojo.

  Parámetros:

  val El cerrojo (lock) es un conjunto de este valor. Usado para el debugging, debe ser distinto de cero.

- VOID InitLock(PIN_LOCK *lock)

  Inicializa el cerrojo como libre.

- INT32 ReleaseLock(PIN_LOCK *lock)

  Liberar el cerrojo.

  Devuelve:

  El valor anterior del cerrojo. Usado para debugging.

B.13. KNOB: Manejo de la línea de comandos

Descripción detallada

Knobs automatiza el "parsingz la gestión de las llamadas de la linea de comandos. Una linea de comandos contiene llamadas para Pin, la herramienta, y la aplicación. El código de "parsing"de los knobs sabe como separarlas (es un analizador sintáctico de la línea de comandos).

Tipos Enumerados

- enum LEVEL_BASE::KNOB_MODE { KNOB_MODE_INVALID, LEVEL_BASE::KNOB_MODE_COMMENT, LEVEL_BASE::KNOB_MODE_WRITEONCE, LEVEL_BASE::KNOB_MODE_OVERWRITE, LEVEL_BASE::KNOB_MODE_ACCUMULATE, LEVEL_BASE::KNOB_MODE_APPEND, KNOB_MODE_LAST }

B.14. Funciones

- BOOL LEVEL_PINCLIENT::ParseCommandLine (int xargc, CHAR **xargv)
- LOCALFUN BOOL LEVEL_PINCLIENT::PIN_ParseCommandLine (int xargc, CHAR **xargv)
- LEVEL_BASE::KNOB_BASE::KNOB_BASE (const string &myname, const string &myfamily, const string &mydefault, const string &mypurpose, KNOB_MODE mymode=KNOB_MODE_WRITEONCE)
- int LEVEL_BASE::KNOB_BASE::Compare (const KNOB_BASE &k2) const
- STATIC VOID LEVEL_BASE::KNOB_BASE::CheckAllKnobs ()
- STATIC UINT32 LEVEL_BASE::KNOB_BASE::NumberOfKnobs ()
- STATIC VOID LEVEL_BASE::KNOB_BASE::DisableKnobFamily (const string &myfamily)
- STATIC VOID LEVEL_BASE::KNOB_BASE::EnableKnobFamily (const string &myfamily)
- STATIC VOID LEVEL_BASE::KNOB_BASE::DisableKnob (const string &myname)
- STATIC VOID LEVEL_BASE::KNOB_BASE::EnableKnob (const string &myname)
- STATIC KNOB_BASE * LEVEL_BASE::KNOB_BASE::FindKnob (const string &name)
- STATIC KNOB_BASE * LEVEL_BASE::KNOB_BASE::FindFamily (const string &name)
- STATIC KNOB_BASE * LEVEL_BASE::KNOB_BASE::FindEnabledKnob (const string &name)
- STATIC string LEVEL_BASE::KNOB_BASE::StringKnobSummary ()

Documentación de tipos Enumerados

enum LEVEL_BASE::KNOB_MODE Indica los modos de knob soportados.

Valores de enumeración:

KNOB_MODE_COMMENT	Comentario para la familia de knobs
KNOB_MODE_WRITEONCE	Valor simple, escritura simple
KNOB_MODE_OVERWRITE	Valor simple, sobreescritura
KNOB_MODE_ACCUMULATE	Valor simple, actualización
KNOB_MODE_APPEND	Lista de valores, añadir al final

Documentación de las Funciones

- VOID LEVEL_BASE::KNOB_BASE::CheckAllKnobs( )

  Busca duplicados entre los knobs declarados

- int LEVEL_BASE::KNOB_BASE::Compare(const KNOB_BASE &k2)

  Devuelve:

  TRUE si dos knobs son idénticos.

- VOID LEVEL_BASE::KNOB_BASE::DisableKnob(const string &myname)

  Invalida todas las opciones dentro de un knob dado.

  Parámetros:

  myname El knob a invalidar.

- VOID LEVEL_BASE::KNOB_BASE::DisableKnobFamily(const string &myfamily)

  Invalida todas las opciones dentro de una familia de knobs dados.

  Parámetros:

  myfamily La familia a invalidar.

- VOID LEVEL_BASE::KNOB_BASE::EnableKnob(const string &myname)

  Habilita un knob en particular

  Parámetros:

  myname El knob a habilitar

- VOID LEVEL_BASE::KNOB_BASE::EnableKnobFamily(const string &myfamily)

  Habilita todas las opciones dentro de una familia dada de knobs.

  Parámetros:

  myfamily La familia a habilitar

- KNOB_BASE * LEVEL_BASE::KNOB_BASE::FindEnabledKnob(const string & myname)

  Busca un knob que actualmente no esté desactivado.

  Parámetros:

  myname El nombre del knob activado a buscar.

- KNOB_BASE * LEVEL_BASE::KNOB_BASE::FindFamily(const string &family)

  Busca una familia de knobs

  Parámetros:

  family La familia de knobs a localizar.

- KNOB_BASE * LEVEL_BASE::KNOB_BASE::FindKnob(const string & myname)

  Localiza un knob de la lista de knobs declarados.

  Parámetros:

  myname El nombre del knob a localizar.

- LEVEL_BASE::KNOB_BASE::KNOB_BASE(const string &myname, const string &myfamily, const string &mydefault, const string &mypurpose, KNOB_MODE mymode = KNOB_MODE_WRITEONCE)

  Crea un nuevo knob

  Parámetros:

  myname Nombre del knob

  myfamily Familia a la que pertenece el knob

  mydefault El valor por defecto al que se inicializa el knob.

  mypurpose Un string que explica el propósito del knob

  mymode KNOB_MODE

- UINT32 LEVEL_BASE::KNOB_BASE::NumberOfKnobs()

  Devuelve:

  El número total de knobs declarados

- BOOL ParseCommandLine(int xargc, CHAR ** xargv)

  Despreciable. Chequea el valor devuelto por PIN_Init().

- LOCALFUN BOOL PIN_ParseCommandLine(int xargc, CHAR ** xargv)

  Analiza sintácticamente la linea de comandos (parsing), ajustando los knobs de manera apropiada.

- string LEVEL_BASE::KNOB_BASE::StringKnobSummary()

  Imprime un sumario de todos los knobs declarados.

B.15. Información sobre asignación de memoria

Funciones

- INT32 LEVEL_PINCLIENT::PIN_NumPinMemoryRangeElements ()
- VOID LEVEL_PINCLIENT::PIN_PinMemoryRanges (VOID *ranges[])

Documentación de Funciones

- INT32 PIN_NumPinMemoryRangeElements()

  Numero de elementos a asignar del array para soportar los rangos de memoria (elementos del vector ranges) de la función PIN_PinMemoryRanges

- VOID PIN_PinMemoryRanges(VOID * ranges[])

  Función usada para encontrar toda la memoria asignada por pin . Se rellena en un array de una dimensión (vector) con pares de direcciones de memoria (dirección de comienzo, dirección de fin).

Apéndice C
Palabras clave

1. Cache
2. Hardware adaptativo
3. Instrumentacion
4. Pin
5. Simulador dinámico
6. Consumo de energía
7. Tasa de fallos
8. Monitorización
9. Trazas
10. Dinero IV